Software Edition
ICP Verlag

# Optimierender Makro-Assembler für Commodore Amiga

Das professionelle Assembler-System für den Commodore Amiga. Bestehend aus Hitex-Editor, Makro-Assembler, symbolischem Debugger, Linker, mehreren Hilfsprogrammen und deutschem Handbuch.

**Dieses Paket ist eines der umfangreichsten und besten Assembler-Systeme für den Commodore Amiga. Es bietet alle Komponenten eines professionellen Entwicklungssystems, kann aber auch von einem Anfänger schnell erlernt werden.**

**Optimierender Makro-Assembler V:**

- Kennt 68000, 68010, 68020, 68030, 68040, 68881, 68882, 68851
- Unterstützt Source-Level-Debugging
- Unterstützt objektorientierte Programmierung
- Kompatibel zum Devpac- und Metacomco-Assembler, Zusatzdirektiven
- Kommandozeilen-kompatibel zum Metacomco, Zusatzoptionen
- Läuft auf allen Amiga-Betriebssystemversionen
- Ab Workbench 1.3 resident haltbar
- Über 300 Fehlermeldungen und Warnungen
- Listingerzeugung und -formatierung wie Metacomco, Zusatzdirektiven
- Direktiven sind redefinierbar, daher mögliche Cross-Assemblierung für andere Prozessoren
- Beliebig viele Include-Directories, ein Default-Include-Directory
- Zeilenlänge bis zu 255 Zeichen, freies Zeilenformat
- Symbollänge bis zu 255 Zeichen, beliebig viele Symbole
- Beliebige Makrogröße und -verschachtelung
- 36 Makroparameter plus Null-Parameter, Weiterzähl-Parameter und Spezial-Makro-Befehle
- Beliebige Includefile-Verschachtelung
- Beliebige Verschachtelung von IF-Direktiven
- Bis zu 255 Sektionen
- Ausdrucksschachtelung bis zu 32 Klammern
- Stellenweise oder global ein- und ausschaltbarer Optimierer erzeugt hochoptimierten Code
- Hochoptimierter Mehrpass-Modus
- Hohe Assemblier-Geschwindigkeit: ca. 63000-74000 Zeilen pro Minute, Peak-Geschwindigkeit von 200000 Zeilen pro Minute (NOPs)
- Wenn in den Editor integriert: direktes Assemblieren aus den Editorpuffern, direktes Fehleranspringen und Assemblieren in den Speicher
- Kann direkt ausführbare Programme erzeugen
- Vorassemblierte Include-Files möglich
- »Small data«-Modell
- Geringer Speicherbedarf

**Symbolischer Debugger DBug V1.1**

- Symbolischer Multiwindow-Debugger mit Unterstützung aller Prozessoren
- Source-Level-Unterstützung bietet Abarbeitung direkt im Quelltext
- Für alle Amiga-Modelle und alle Betriebssysteme ab Kickstart 1.2
- Debuggen beliebiger Tasks oder Programme
- Einzelschrittmodus, schnelles Ausführen von Unterprogrammen
- Breakpoints, Speicher-Breakpoints, zeitweilig ausschaltbar
- Anzeige beliebiger Speicherbereiche in verschiedenen Formaten, integrierter Assembler und Disassembler für alle Prozessoren
- History-Funktion mit beliebiger Tiefe
- Logfile zum Protokollieren
- Fontsensitiv: verwendet den eingestellten System-Defaultfont

**Hitex-Editor:**

- Arbeitet auf allen Amiga-Modellen ab Kickstart 2.0, volle 2.0-, ECS-, AA-, AGA- und sonstige Unterstützung
- ARexx-Unterstützung
- Frei konfigurierbar und fontskalierend
- Beliebig viele Puffer, Textgröße nur durch verfügbaren Speicher begrenzt
- Beliebige, persistente Marken
- Fehlerlisten-Unterstützung für alle Compiler und Assembler
- Eingebautes Make-Utility
- Geeignet für alle Sprachen von Assembler über C bis Cluster
- Workbench- und Shell-fähig
- Steuerung über Tastatur und Maus
- Wählbare Backup-Modi
- Komfortables Blocklegen per Maus, umfangreiche Blockoperationen und permanent sichtbarer Block
- Schnelle Such- und Ersetzoperation mit vielen Optionen
- Fließtextunterstützung
- Automatisch ladbare Voreinstellungsdatei

Hardwarevoraussetzungen: Jedes Amiga-Modell ab 512 KByte RAM. Kickstart 1.2, 1.3 oder OS 2.0. Volle ECS-, AA-, AGA- und sonstige Unterstützung.

**Vertrieb: ICP GmbH & Co. KG, Postfach 1143, 85591 Vaterstetten/München**
**Ein Tochterunternehmen der GONG-Gruppe**

# Optimierender Makro-Assembler

# Version 3.0

Handbuch und Programme sind Copyright 1994

Dietmar Heidrich, Blenderstraße 60
D-28279 Bremen

Stand: 11. Oktober 1994

Vertrieb und Support:
ICP-Innovativ Computer Presse GmbH & Co. KG
Wendelsteinstraße 3
D-85591 Vaterstetten

## Vorwort

Gratulation zum Erwerb des Optimierenden-Makro-Assemblers V3.0. Sie besitzen nun eines der leistungfähigsten Assemblerpakete für die Commodore-Amiga-Familie. Bevor Sie die gelieferten Programme benutzen, sollten Sie die Handbücher genau lesen, um Irrtümern und Ärgernissen vorzubeugen.

Alle Programme arbeiten mit speichererweiterten Maschinen und in Verbindung mit allen Prozessoren der M68000-Familie. Ab Workbench 1.3 können Sie alle Programme resident halten, nur Editor und Debugger dürfen dann nicht gleichzeitig von mehreren Prozessen aus benutzt werden. Die übrigen Programme verkraften dies problemlos. Das Paket arbeitet auch ab dem Betriebssystem 2.0 korrekt.

Ganz besonders möchte ich Herrn Dieter Reitzner danken, ohne dessen beraterische Tätigkeit und Verbesserungsvorschläge das Paket nicht in dieser Form verfügbar gewesen wäre.

**Alle Programme und Dateien sowie die Handbücher sind Copyright 1994 bei Dietmar Heidrich, Blenderstraße 60, D-28279 Bremen.**

**Jede Art der Reproduktion, Verbreitung oder Verteilung, ganz oder teilweise, der Handbücher oder Programme oder Dateien ist ausdrücklich laut Lizenzbedingungen verboten. Zuwiderhandlungen werden juristisch verfolgt.**

Sollten Sie Fehler in einem der Programme entdecken, stellen Sie bitte fest, ob und unter welchen Bedingungen der Fehler reproduzierbar ist. Versuchen Sie, dieses genau zu beschreiben und erwähnen Sie, mit welchen Komponenten Ihr Rechner ausgestattet ist. Melden Sie den Fehler an:

**ICP-Innovativ Computer Presse GmbH & Co. KG**
**Wendelsteinstraße 3, D-85591 Vaterstetten**

Falls Sie nun das Vertrauen in OMA verloren haben sollten, lassen Sie sich beruhigen: Das gesamte Assemblerpaket wurde mit sich selber erstellt, assembliert und gelinkt. Die Programme befinden sich nun schon seit einigen Jahren erfolgreich im Einsatz, und es wurden alle bekannten Fehler eliminiert.

Dieses Handbuch wurde mit dem HiTex-Editor Version 3.0 erstellt und mit Hilfe von TEX gesetzt.

Amiga, Amiga-DOS, Workbench, Intuition, GadTools und ASL sind Warenzeichen von Commodore-Amiga, Inc.
Metacomco ist ein Warenzeichen von Metacomco plc.
Devpac ist ein Warenzeichen von HiSoft.

# Inhalt

**1 Einführung** **9**
1.1 Installation . . . . . 9
1.2 Tutorium . . . . . 10

**2 HiTex-Editor V3.0** **13**
2.1 Einführung . . . . . 13
2.2 Starten des Editors . . . . . 15
2.2.1 Start von einer Shell . . . . . 15
2.2.2 Start von der Workbench . . . . . 16
2.2.3 Nach dem Start . . . . . 17
2.2.4 Startprobleme ? . . . . . 17
2.3 Textpuffer . . . . . 18
2.3.1 Textpufferfenster . . . . . 18
2.3.2 Icons . . . . . 19
2.3.3 Marken . . . . . 20
2.3.4 Texteingabe . . . . . 21
2.3.5 Der Block und die Maus . . . . . 22
2.3.6 Privater ARexx-Port . . . . . 24
2.4 Hilfsfenster und Requester . . . . . 25
2.4.1 Gemeinsamkeiten und Unterschiede . . . . . 25
2.4.2 Das Pufferlistenfenster . . . . . 26
2.4.3 Das Markenfenster . . . . . 26
2.4.4 Der Filerequester . . . . . 26
2.4.5 Das Findfenster . . . . . 28

2.4.6 Die Preferences-Requester . . . 28
2.4.7 Der Print-Requester . . . 28
2.4.8 Der Assemble-Requester . . . 28
2.4.9 Der Modified-Project-Requester . . . 28
2.5 Tastatur-Funktionen und Menüs . . . 29
2.6 Such-Funktion . . . 31
2.6.1 Das Findfenster . . . 31
2.6.2 Wildcards . . . 32
2.6.3 Besonderheiten . . . 33
2.7 ARexx . . . 34
2.7.1 ARexx-Ports . . . 34
2.7.2 ARexx-Details . . . 35
2.7.3 Probleme mit ARexx . . . 35
2.7.4 ARexx-Befehle zur Pufferverwaltung . . . 36
2.7.5 ARexx-Befehle zur Blockverwaltung . . . 39
2.7.6 ARexx-Befehle zur Cursorbewegung . . . 41
2.7.7 ARexx-Befehle zur Markenverwaltung . . . 43
2.7.8 ARexx-Befehle zur Fensterverwaltung . . . 44
2.7.9 ARexx-Befehle für die Preferences . . . 46
2.7.10 ARexx-Steuerbefehle und Makros . . . 48
2.7.11 ARexx-Befehle zum Suchen/Ersetzen . . . 50
2.7.12 ARexx-Befehle zur Statusinformation . . . 51
2.7.13 ARexx-Befehle zur Textänderung . . . 53
2.7.14 ARexx-Befehle für Assembler und Fehlerliste . . . 56
2.8 Das Make-Utility . . . 58
2.8.1 Beschreibung . . . 58
2.8.2 Der residente Assembler . . . 60
2.8.3 Assemblieren in den Speicher . . . 61
2.9 Die Preferences . . . 62
2.9.1 Der Standard-Preference-Requester . . . 62
2.9.2 Der Screen-Preference-Requester . . . 63
2.9.3 Die Preferencedatei . . . 64
2.10 Konfiguration des Editors . . . 67



2.10.1 Menükonfiguration . . . 67
2.10.2 Tastenkonfiguration . . . 68
2.10.3 Ausgeschaltete Menüs und Checkmarks . . . 70

**3 Optimierender Makro-Assembler V2.1 71**
3.1 Einführung . . . 71
3.2 Aufruf des Assemblers . . . 74
3.2.1 Aufruf . . . 74
3.2.2 Fast-Modus und Mehrpass-Modus . . . 77
3.2.3 Beispielaufrufe . . . 77
3.2.4 Anhalten des Assemblers . . . 78
3.3 Die Zeilensyntax . . . 79
3.3.1 Die Syntax . . . 79
3.3.2 Lokale Symbole . . . 80
3.4 Ausdrücke . . . 81
3.4.1 Symbolsyntax und Konstanten . . . 81
3.4.2 Rechenregeln . . . 82
3.4.3 Fließkomma-Ausdrücke . . . 83
3.5 Adressierungsarten . . . 85
3.5.1 Normale Adressierungsarten . . . 85
3.5.2 Adressierungen ab MC68020 . . . 87
3.5.3 Spezialregister . . . 88
3.5.4 Adressierungs-Schablonen . . . 89
3.6 Die Direktiven . . . 90
3.6.1 Steuerungsdirektiven . . . 90
3.6.2 Zuweisungsdirektiven . . . 98
3.6.3 Objektorientierte Direktiven . . . 104
3.6.4 Listingdirektiven . . . 105
3.6.5 Bedingte Assemblierung . . . 107
3.6.6 Makros . . . 110
3.6.7 Makro-Parameter, -Spezialbefehle und -Beispiele . . . 111
3.7 Konvertierung anderer Quelltexte, Inkompatibilitäten . . . 118
3.7.1 Inkompatibilitäten zum Metacomco-Assembler Version 11.00 . . 118

3.7.2 Inkompatibilitäten zum Devpac-Assembler Version 2, Oktober 1988 119
3.8 Beginn einer Sektion . . . 121
3.9 Optimierungen und Konvertierungen . . . 122
3.10 Fehlermeldungen . . . 124
3.11 Verkürzte Adressierung . . . 125
3.12 Das Listingfile . . . 128
3.12.1 Das Listing . . . 128
3.12.2 Cross-Referenz und Symboltabelle . . . 129
3.13 Vorassemblierte Includefiles . . . 131
3.14 Source-Level-Debugging . . . 132
3.15 32-Bit-Prozessoren und 32-Bit-Probleme . . . 133

**4 Symbolischer Debugger DBug V1.1 135**
4.1 Einführung . . . 135
4.2 Eingabe-Elemente . . . 137
4.2.1 Anklicken von Daten . . . 137
4.2.2 Fenster-Eigenschaften . . . 137
4.2.3 Requester . . . 138
4.2.4 Integer-Ausdrücke . . . 139
4.2.5 Fließkomma-Ausdrücke . . . 140
4.3 Fenstertypen . . . 142
4.3.1 Das Speicherfenster . . . 142
4.3.2 Das Statusfenster . . . 144
4.3.3 Das BigStatus-Fenster . . . 144
4.3.4 Das FPU-Statusfenster . . . 145
4.3.5 Das Hunk-Fenster . . . 145
4.3.6 Das Symbol-Fenster . . . 145
4.3.7 Das Breakpoint-Fenster . . . 146
4.3.8 Das History-Fenster . . . 146
4.3.9 Das Text-Fenster . . . 147
4.3.10 Das Source-Level-Fenster . . . 148
4.4 Debuggen von Tasks . . . 149
4.4.1 Abfangen der nächstgestarteten Task . . . 149



4.4.2 Abfangen einer bereits laufenden Task . . . 149
4.4.3 Abfangen einer abgestürzten Task . . . 149
4.4.4 Unterbrechen des Abfangens . . . 150
4.4.5 Nach dem Abfangen . . . 150
4.4.6 Beenden der Task . . . 151
4.5 Die Menüs . . . 152
4.5.1 Das Project-Menü . . . 152
4.5.2 Das Open-Menü . . . 153
4.5.3 Das Trace-Menü . . . 153
4.5.4 Das Break-Menü . . . 155
4.5.5 Das Command-Menü . . . 156
4.5.6 Das Window-Menü . . . 157
4.5.7 Das Order-Menü . . . 158
4.5.8 Das Source-Menü . . . 159
4.5.9 Das Address-Menü . . . 159
4.5.10 Das Data-Menü . . . 160

**5 DLink V1.2 163**
5.1 Einführung . . . 163
5.2 Aufruf des Linkers . . . 164
5.3 Short-Modus . . . 167
5.4 Libraries . . . 168

**6 Hilfsprogramme und die Library OMA.lib 169**
6.1 Make . . . 169
6.2 Absolute . . . 169
6.3 Strip . . . 170
6.4 Cmp . . . 170
6.5 CLS . . . 170
6.6 Das Include-Directory . . . 171
6.7 Das Examples-Directory . . . 171
6.8 Das Libraries-Directory und OMA.lib . . . 171

**7 Objektorientierte Programmierung** **177**
7.1 Einführung in OOP . . . 177
7.1.1 Motivationsbeispiel . . . 177
7.1.2 OOP-Grundbegriffe . . . 178
7.1.3 Methoden . . . 179
7.2 OOP-Struktur bei OMA . . . 179
7.2.1 Klassenhierarchie und ClassObject . . . 180
7.2.2 Erzeugen eines Objektes . . . 180
7.2.3 Methodenaufrufe . . . 181
7.2.4 Vernichtung von Objekten . . . 181
7.3 Implementierung von Klassen . . . 181
7.3.1 Schritt 1: Objektstruktur deklarieren . . . 182
7.3.2 Schritt 2: Methoden deklarieren . . . 182
7.3.3 Schritt 3: Klassencontainer anlegen . . . 183
7.3.4 Schritt 4: Methoden implementieren . . . 183
7.3.5 Schritt 5: Klassen registrieren . . . 185
7.4 OOP-Beispiel . . . 185
7.5 Benennungskonventionen . . . 186

# Kapitel 1

# Einführung

## 1.1 Installation

Für die Installation setze ich voraus, daß Sie den Amiga mit dem CLI bedienen können und sich dort einigermaßen auskennen. Sollte dies nicht der Fall sein, dann lesen Sie bitte Ihr Bedienerhandbuch zum Amiga durch, speziell den Amiga-DOS-Teil. Vor allen anderen Aktionen sollten Sie eine Kopie von Ihrer Original-Assembler-Diskette machen. Arbeiten Sie nicht mit dem Original, sondern immer mit der Kopie.

1. Schieben Sie die Assembler-Diskette in ein Laufwerk.
2. Lesen Sie mit einem Editor die auf der Diskette enthaltene Datei LiesMich oder lassen Sie sie auf dem Terminal mittels
   *Type "OMA V3.0:LiesMich"*
   ausgeben. Da das Assemblerpaket ständig überarbeitet wird, enthält diese Datei aktuelle Informationen über z.B. Fehlerkorrekturen oder Erweiterungen, die nicht in den Handbüchern stehen.
3. Erzeugen Sie auf Ihrer Systemdiskette (evtl. eine Festplatten-Partition) ein Directory namens ACom durch den CLI-Befehl
   *Makedir SYS:ACom*
4. Kopieren Sie den Inhalt der Assembler-Diskette in das gerade erzeugte Directory. Dazu können Sie die Batchdatei der Assembler-Diskette benutzen, da auf der Assembler-Diskette einige Directories und Dateien vorhanden sind, die Sie nicht auf Ihrer Systemdiskette benötigen:
   *Execute "OMA V3.0:Kopiere.batch" SYS:ACom*
   Achten Sie darauf, daß Sie genug Platz auf Ihrer Systemdiskette haben, denn sonst bekommen Sie eine Systemmeldung. Schaffen Sie in dem Fall genug Platz und wiederholen Sie den obigen Befehl.
5. Nehmen Sie mit einem Editor folgende Zeilen in Ihre Startup-Sequenz im DEVS:-Directory auf (Reihenfolge nicht ändern):

*Assign ACOM: SYS:ACom*
*Assign AINC: ACOM:Include*
*Assign ALIB: ACOM:Libraries*
*Path ACOM: add*
Den letzten Befehl können Sie natürlich in einem evtl. bereits vorhandenen Path-Befehl unterbringen.

6. Setzen Sie Ihren Rechner zurück und booten Sie neu. Nun können Sie das Assemblerpaket benutzen und die Programme starten. Falls Sie die Workbench 1.3 besitzen und die Shell benutzen, sollten Sie dieser (evtl. in der Datei S:Shell-Startup) noch
*Alias ED Run ACOM:Editor []*
befehlen, damit Sie unter der Shell nur noch ED aufrufen müssen, um den Editor als Hintergrundbefehl ablaufen zu lassen.

Bitte beachten Sie, daß bei Benutzung der Workbench mit dem Editor das Zuweisen des ACOM:-Directories notwendig wird, da der Editor Icons zu den Texten abspeichert, deren Default-Tool »ACOM:Editor« lautet. Daher sollten Sie den oben gezeigten Weg befolgen.

## 1.2 Tutorium

Dieses Kapitel ist als (sehr) kurze Einführung in die Bedienung des Assemblerpaketes gedacht. Sie werden sich dadurch nicht ersparen, die Handbücher komplett zu lesen, da die Programme viel zu komplex sind, als daß es möglich wäre, ihren Funktionsumfang auf ein paar Seiten zu beschreiben.

Gehen Sie in das CLI oder in eine Shell und rufen Sie den Editor per `run Editor` bzw. `run ACOM:Editor` (siehe obige Installation) auf. Das Programm startet und wartet auf Ihre Befehle. In der Menüleiste finden Sie ganz rechts das Assembler-Menü. Wählen Sie in diesem Menü den Punkt Load Assembler. Es erscheint der File-Requester des Editors. Wählen Sie nun den Assembler zum Laden aus, indem Sie seinen Namen eingeben oder doppelt anklicken. Nun können Sie Assemblerprogramme direkt vom Editor aus übersetzen lassen. Der Aufruf dazu befindet sich ebenfalls im Assembler-Menü unter Assemble. Sie möchten nun ein Beispiel ausprobieren (doch, doch, das möchten Sie!). Geben Sie dem aktuellen Textpuffer erst einmal einen Namen, indem Sie Rename buffer aus dem Buffer-Menü auswählen, beim dann erscheinenden File-Requester den Namen `beispiel.asm` eintippen und <Return> drücken. Geben Sie jetzt folgenden Text ein:

```
      moveq  #0,d0
1$    addq.l #1,d0
      cmp.l  #1000000,d0
      blo.s  1$
      moveq  #0,d0      ; Rueckgabe an das DOS
      rts
      END
```



Das Programm zählt einfach bis zu einer Million. Rufen Sie nun Assemble auf. Der Assemble-Requester erscheint mit einer Anzahl Gadgets und dem gleich aktivierten Objektfilenamen. Drücken Sie nicht <Return>, da sonst der Assemblerlauf gestartet würde, sondern überprüfen Sie erst einmal die Gadgets: »Fast Mode«, »Statistics«, »Optimize«, »Object file« nicht, »Case distinct«, »Warnings« und »Into memory« sollten gewählt sein. Wenn Sie dies einmal eingestellt haben, brauchen Sie bei weiteren Assemblerläufen keine Änderung zu machen. Klicken Sie nun auf das »Ok«-Gadget und sehen Sie, was passiert.

Der Editor schaltet auf die Workbench, öffnet ein Fenster, und der Assembler beginnt zu arbeiten. Wenn er fertig ist, bekommen Sie eine entsprechende Meldung. Drücken Sie nun wie verlangt die Return-Taste. Falls beim Assemblieren ein Fehler auftrat, wird jetzt die fehlerhafte Zeile angesteuert, und der Editor druckt die Fehlermeldung auf die unterste Zeile (die unter dem durchgezogenen Strich). Sie können nun das fertige Programm ausführen lassen, indem Sie den Menüpunkt Run wählen. Als Alternative können Sie das Make-Utility des Editors benutzen, welches Ihnen nicht nur erlaubt, große, linkbare Programme zu pflegen, sondern sie auch gleich zu starten. Dazu müssen Sie aber das Programm z.B. in die RAM-Disk assemblieren bzw. linken lassen und wie unter dem CLI starten.

Da Sie sich gerade im Editor befinden, sollten Sie ihn jetzt einmal ausprobieren. Sie können nichts kaputt machen, solange Sie keine Texte über das Project-Menü speichern. Testen Sie, wie die Cursor-Tasten arbeiten, speziell in Zusammenhang mit <Shift>, <Alt> usw. Der Editor wird nämlich für Ihre Arbeit mit OMA die Plattform bilden und eignet sich im Gegensatz zu Assembler und Linker hervorragend zum Eingewöhnen durch Herumspielen. Falls Sie etwas nicht verstehen, können Sie das Stichwortverzeichnis im Editor-Handbuch benutzen.

# Kapitel 2

# HiTex-Editor V3.0

## 2.1 Einführung

Das wichtigste Werkzeug der Programmentwicklung ist ein leistungsfähiger, komfortabler Editor. Dabei ist es egal, in welcher Sprache entwickelt wird, denn die meiste Zeit wird beim Editieren benötigt. Mit dem HiTex-Editor Version 3.0 haben Sie nun einen universellen Texteditor in der Hand, der flexibel genug ist, sich Ihren Bedürfnissen anzupassen. Er wurde so konzipiert, daß Ihnen bei der anstrengenden Programmiertätigkeit möglichst keine Steine in den Weg gelegt werden, schließlich möchten Sie Programme schreiben und sich nicht über unzulängliche Werkzeuge ärgern.

Der HiTex-Editor 3.0 stammt von den Editoren des Optimierenden Makro-Assemblers ab. Er ist mit dem User-Interface-Style-Guide von Commodore konform und unterstützt beliebige Fonts. Er kann auf einem Public-Screen oder einem eigenen Screen laufen. Zu seinen Highlights gehören eine freie Konfigurierbarkeit von Tastaturkommandos und Menüs sowie eine umfangreiche Unterstützung von ARexx, der seit Kickstart 2.0 jedem Amiga beiliegenden Skript-Programmiersprache. Mit Hilfe von ARexx ist es möglich, auf der vom Editor gelieferten Funktionalität aufzubauen und Funktionen zu realisieren, die immens komplexe Operationen auf den Texten durchführen.

Der HiTex-Editor verarbeitet und erzeugt Texte im ASCII-Format. Dabei ist folgendes zu beachten:

- Eine Zeile darf bis zu 254 Zeichen lang sein. Sie endet immer mit einem Linefeed (Code $0A).
- Bei eingelesenen Fremd-Dateien wird ein Zeilenende nur an einem Linefeed erkannt. Carriage-Return (Code $0D) alleine reicht nicht aus. Überlange Zeilen spaltet der Editor in mehrere Zeilen auf und meldet diese Aktion. Die Kombination $0D-$0A bewirkt nur einen Zeilenwechsel, das Carriage-Return bleibt allerdings sichtbar.

Ich empfehle Ihnen, dieses Handbuch am Rechner durchzuarbeiten und die Funktionen des Editors gleich auszuprobieren. Dies wird Ihnen helfen, die Arbeitsweise des Editors zu verstehen, und beugt Mißverständnissen vor. Im Handbuch wird bei Menü- und Tastenbelegungen von der Standardkonfiguration ausgegangen. Da der Editor jedoch frei konfigurierbar ist, können Sie die Befehle so auf Menüs und Tasten legen, wie Sie es Ihnen gefällt (siehe auch Kapitel 2.10 über die Editor-Konfiguration).

Hier noch eine kleine Zusammenfassung der Leistungsmerkmale:

- arbeitet auf allen Amiga-Modellen ab Kickstart 2.0, volle 2.0-, ECS-, AA-, AGA- und sonstige Unterstützung
- ARexx-Unterstützung
- frei konfigurierbar und fontskalierend
- beliebig viele Puffer, Textgröße nur durch verfügbaren Speicher begrenzt
- beliebige, persistente Marken
- Fehlerlisten-Unterstützung für alle Compiler und Assembler
- eingebautes Make-Utility
- geeignet für alle Sprachen von Assembler über C bis Cluster etc.
- Workbench- und Shell-fähig
- Steuerung über Tastatur und Maus
- wählbare Backup-Modi
- komfortables Blocklegen per Maus, umfangreiche Blockoperationen und permanent sichtbarer Block
- schnelle Such- und Ersetzoperationen mit vielen Optionen
- Fließtextunterstützung
- automatisch ladbare Voreinstellungsdatei



## 2.2 Starten des Editors

Der HiTex-Editor läßt sich sowohl von einer Shell als auch von der Workbench aus starten. Dabei können jeweils beim Start gleich Dateien als Puffer eingeladen werden.

### 2.2.1 Start von einer Shell

Die Syntax für den Shell-Start lautet:

`Editor FILES/M SCREEN/K PORTNAME/K PREFS/K STARTUP/K SETUP/K`

Dabei haben die Argumente genau die Standardform wie unter Amiga-DOS üblich (siehe auch Amiga-DOS-Handbuch). Die einzelnen Argumenttypen gliedern sich wie folgt auf:

**FILES/M** Es dürfen soviele Dateinamen angegeben werden wie gewünscht. Jede angegebene Datei wird als Puffer gleich beim Start in den Editor geladen. Die Dateinamen dürfen auch die üblichen Amiga-DOS-Patterns haben, z.B. lädt `Editor #?.txt ReadMe` die Datei namens ReadMe sowie alle Dateien, die die Endung ».txt« besitzen. Der Dateiname muß in Amiga-DOS-Manier wie gewohnt angegeben werden, d.h., der String »a**b ist c« repräsentiert die Datei a*b ist c. Die geladenen Dateien haben als Heimatverzeichnis zuerst das aktuelle Verzeichnis der Shell.

**SCREEN/K** Der Editor kann auf dem Workbench-Screen oder einem eigenen Screen laufen. Hiermit wird der Screenname angegeben, auf dem der Editor seine Fenster öffnen soll. Falls ein Public-Screen mit diesem Namen existiert, läuft der Editor auf diesem, ansonsten öffnet er einen eigenen Screen. Siehe auch Kapitel 2.9 über die Preferences.

**PORTNAME/K** Dies ist das Präfix (d.h. der Anfang) des Namens des ARexx-Hauptports, der verwendet werden soll. Jedes ARexx-unterstützende Programm benötigt einen Port, an dem Meldungen ankommen. Wird dieses Argument nicht angegeben, dann wird »HITEX.AREXX.« als Präfix verwendet. Der Editor bekommt als Hauptportnamen denjenigen, der aus dem Präfix und der Anzahl bereits laufender HiTex-Editoren gebildet wird. Wenn z.B. bereits zwei HiTex-Editoren auf dem Rechner laufen, bekommt der dritte den Hauptportnamen »HITEX.AREXX.3«. Ein mit PORTNAME angegebenes Präfix könnte hier z.B. zum Namen »MeinEditor3« führen. Bitte beachten Sie, daß nach Commodore-Empfehlung der ARexx-Portname vollständig in Großbuchstaben verfaßt sein sollte und keine Leerzeichen enthalten darf.

**PREFS/K** Mit diesem Argument kann eine Preferencedatei angegeben werden, aus der der Editor beim Start seine Standardeinstellungen liest. Welche Standardeinstellungen existieren, entnehmen Sie bitte dem Kapitel über die Preferences. Falls dieses Argument nicht angegeben ist, versucht der Editor, eine Datei namens HiTex.prefs aus dem aktuellen Verzeichnis zu holen. Gelingt dies nicht, probiert er es noch einmal mit der Datei ACOM:HiTex.prefs. Falls diese auch nicht existiert, dann werden die internen Voreinstellungen benutzt.

**STARTUP/K** Dies ist der Name eines ARexx-Programmes, das gleich nach dem Einlesen aller Puffer ausgeführt werden soll (siehe auch Kapitel 2.7 über ARexx). STARTUP kann z.B. dafür verwendet werden, den Editor mit Hilfe eines ARexx-Programmes für bestimmte Editiermodi einzustellen, die vom der normalen Preference-Einstellung abweichen. Wie beim Befehl EXECAREXXMACRO beschrieben, sollte das mit STARTUP angegebene ARexx-Programm in seinem Namen keine Leerzeichen enthalten, da sonst u.U. das Programm nicht gefunden wird (dies ist ein schwerer Nachteil von ARexx). Falls STARTUP nicht angegeben ist, versucht der Editor, eine Datei namens HiTex.hit aus dem aktuellen Verzeichnis auszuführen. Gelingt dies nicht, probiert er es noch einmal mit der Datei ACOM:HiTex.hit. Dies ermöglicht es Ihnen, weitergehende Konfigurationen einzustellen als mit den Preferences und der Setup-Datei (siehe auch Kapitel 2.10 über die Editorkonfiguration).

**SETUP/K** Dies ist eine Setup-Datei wie in Kapitel 2.10 beschrieben. Sie enthält die Konfiguration des Editors für Menüs und Tasten. Genauere Information findet sich im genannten Kapitel.

### 2.2.2 Start von der Workbench

Sie starten den Editor von der Workbench aus, indem Sie einfach sein Icon doppelt anklicken. Möchten Sie, daß der Editor gleich Dateien als Puffer einliest, dann benutzen Sie die erweiterte Auswahl der Workbench. Dabei klicken Sie die Icons der zu ladenden Dateien bei gedrückter Shift-Taste einmal an und starten bei immer noch gedrückter Shift-Taste den Editor durch Doppelanklicken seines Icons. Jeder Puffer hat dann als Heimatverzeichnis das Verzeichnis, aus dem er geladen wurde.

Sie können einem Icon einer Textdatei auch den Editor als Default-Tool angeben (siehe Amiga-Benutzerhandbuch), dann sind Sie aber bei einem Einzelklick auf das Icon nicht in der Lage, mehrere Texte beim Start einzuladen. Der Editor speichert als Default-Tool in seinen Icons die Datei ACOM:Editor (siehe hierzu auch Installation). Als Tooltype-Einträge im Icon des HiTex-Editors gibt es die folgenden:

**SCREEN=<name>** Wie SCREEN beim Shell-Start.

**PORTNAME=<name>** Wie PORTNAME beim Shell-Start.

**PREFS=<name>** Wie PREFS beim Shell-Start, allerdings versucht der Editor, die Datei HiTex.prefs zuerst aus dem Verzeichenis zu laden, aus dem er selber geladen wurde. Erst dann geht es mit ACOM:HiTex.prefs weiter.

**STARTUP=<name>** Wie STARTUP beim Shell-Start.

**SETUP=<name>** Wie SETUP beim Shell-Start.

Ferner unterstützt der Editor Workbench-AppWindows, d.h., wenn ein Icon eines Textes von der Workbench auf einem Editorfenster losgelassen wird, lädt der Editor diesen Text als neuen Puffer ein. Dies funktioniert allerdings nicht, solange ein modales Fenster geöffnet ist (Ausnahme siehe Filerequester).



### 2.2.3 Nach dem Start

Noch vor dem Öffnen des ersten Fensters versucht der Editor, eine Preferencedatei wie unter PREFS beschrieben einzuladen, um die benutzergewünschte Konfiguration herzustellen. Völlig analog dazu wird - wie unter SETUP beschrieben - außerdem eine Datei namens HiTex.setup zuerst über die evtl. angegebene Startoption und dann im aktuellen Verzeichnis des Editors bzw. im ACOM:-Verzeichnis gesucht und geladen, die die Konfiguration der Menüs und Tastaturkommandos bestimmt (siehe auch Kapitel 2.10 über die Editorkonfiguration).

Zudem wird ein Standard-Icon namens ACOM:HiTex.default.info eingelesen, das für Texte ohne Icon abgespeichert wird, sofern das Icon-Abspeichern in den Preferences angeschaltet ist (siehe auch Kapitel 2.3.2).

Falls beim Start keine einzuladenden Dateien angegeben wurden, erzeugt der Editor einen Puffer namens »Unnamed1«, damit sofort gearbeitet werden kann.

Die Tastaturbelegung und -wiederholgeschwindigkeit des Editors ist von den System-Preferences abhängig. Dadurch bekommen Sie automatisch ihre nationale Tastatur.

### 2.2.4 Startprobleme ?

Sollte der Editor beim Start einen nicht-fatalen Alert erzeugen, so war es ihm nicht möglich, irgendwelche Bibliotheken zu öffnen. Um welche Bibliothek es sich handelt, können Sie in den meisten Fällen der Alertnummer entnehmen (siehe Execs C-Headerfile exec/alerts.h bzw. Assembler-Includefile exec/alerts.i). Der Editor funktioniert erst ab der offiziellen 2.0-Version Kickstart 37.175 und benötigt die Bibliotheken DOS, Graphics, Intuition, Icon, Workbench, Diskfont, GadTools und ASL. Weitere Startschwierigkeiten meldet der Editor im Klartext, der allerdings nur beim Start von einer Shell erscheint, nicht beim Workbench-Start.

Sofern nur Textpuffer beim Start geladen wurden, deren WINDOW-Tooltype den Zusatz »HIDDEN« haben, werden keine Textfenster geöffnet. Das kann dazu führen, daß gar kein Fenster offen ist und Sie gleich einen Requester bekommen, der Sie fragt, ob Sie den Editor beenden möchten. Antworten Sie einfach mit »Cancel«, und öffnen Sie damit das Pufferlistenfenster.

## 2.3 Textpuffer

### 2.3.1 Textpufferfenster

Jeder Puffer hat ein eigenes Fenster, das aus mehreren Funktionsbereichen besteht. Es gibt die Titelzeile, ein oder mehrere Editierbereiche des Textes sowie die Statuszeile ganz unten.

#### Titelzeile

Die Titelzeile des Fensters zeigt das Wordwrap-, Overstrike-, TabExpand- und Modifikations-Flag sowie den Dateinamen an. Falls der Wordwrap-Modus eingeschaltet ist (siehe Kapitel 2.3.4 über Wordwrap), erscheint ein »W« als Anzeige dafür. Falls der Overstrike-Modus eingeschaltet ist (siehe Kapitel über 2.3.4 Overstrike), erscheint ein »O« als Anzeige. Für den eingeschalteten TabExpand-Modus erscheint schließlich ein »T«.

Es folgt das Modifikationsflag, das anzeigt, ob der Puffertext verändert wurde. Solange Sie den Text nicht ändern, erscheint hier ein Minuszeichen. Bei der ersten Änderung wechselt die Anzeige in ein Pluszeichen. Der Inhalt eines geänderten Puffers kann nicht versehentlich verloren gehen, da der Editor vor dem Pufferlöschen und vor dem Verlassen prüft, ob es einen geänderten Puffer gibt, und in dem Fall eine Sicherheitsabfrage macht. Erst wenn ein Puffer in eine Datei geschrieben und damit gesichert wird, kehrt das Modifikationsflag zur Minusanzeige zurück.

Nach dem Modifikationsflag wird der Dateiname des Puffers angezeigt. Beim Shell-Start ist dieser mit Dateinamen identisch, den Sie angegeben haben. Beim Workbench-Start und beim normalen Nachladen von Dateien ist dies der Name der Datei in dem jeweiligen Verzeichnis. Bitte beachten Sie, daß Sie auch absolute Pfadnamen für Dateien angeben können, z.B. »RAM:ReadMe«. In dem Fall gehört das Präfix »RAM:« auch zum Dateinamen und wird auch so in der Titelzeile ausgegeben.

#### Editierbereiche

Im Fenster selber werden die Editierbereiche des Puffers angezeigt. Ein Editierbereich ist ein Ausschnitt aus dem Text des Puffers. Sie können über den Befehl SPLITVIEW mehrere Editierbereiche einschalten, die voneinander unabhängig Textausschnitte anzeigen und sich auch überlappen dürfen. Ein Editierbereich ist immer aktiv, er enthält den Cursor. Wenn eine Textänderung im aktiven Bereich gemacht wird, stellen auch die anderen Bereiche diese Änderung sofort dar, sofern sie den geänderten Textbereich anzeigen.

Die Bereiche werden durch Trennzeilen unterteilt. Jede Trennzeile gehört zu dem über ihr stehenden Editierbereich und kann angeklickt und gezogen werden, um die Größe ihres Editierbereiches zu ändern. Es ist nicht möglich, einen Editierbereich kleiner als eine Textzeile zu machen. Die Höhe der Trennzeilen ist über die Preferences einstellbar.



Anklicken einer Trennzeile aktiviert auch ihren Editierbereich. Die Größe eines Editierbereiches ist auch über die Befehle GROWVIEW und SHRINKVIEW einstellbar.

Zwischen den Editierbereichen kann durch Hineinklicken oder über die Befehle NEXTVIEW bzw. PREVVIEW umgeschaltet werden.

Jeder Editierbereich hat seinen eigenen Scrollbar. Dessen Knopfhöhe entspricht dem Anteil der im Editierbereich dargestellten Zeilen an der gesamten Zeilenanzahl des Puffertextes, und seine Knopfposition entspricht immer der Position des Textausschnittes innerhalb des Puffertextes. Wenn Sie oberhalb bzw. unterhalb des Knopfes klicken, blättert der Editor im Text eine Seite zurück bzw. vor. Durch Ziehen des Knopfes können Sie einen beliebigen Textausschnitt anwählen. Der Editor zeigt dabei immer den dazugehörigen Textausschnitt an.

Ein Editierbereich kann mit dem Befehl DELETEVIEW entfernt werden. Der freigewordene Platz wird von seinen Nachbarn eingenommen. Es ist bei den Preferences des Editors einstellbar, welcher Font zur Textanzeige in den Editierbereichen verwendet wird.

Beachten Sie bitte bei Cursorbewegungen, daß abhängig von der Größe des Editierbereiches das Scrollen bereits früher beginnen kann. Der Editor hat jeweils oben, unten, links und rechts einen Bereich, den man normalerweise nicht erreicht, ohne das Scrollen auszulösen. Dieser Bereich wird als scrollsensitiver Bereich bezeichnet und bietet den Vorteil, daß Sie auch noch ein bißchen von dem Text sehen, der als nächstes vom Cursor erreicht wird.

#### Statuszeile

Die Statuszeile befindet sich unter dem letzten Editierbereich. Sie zeigt für den aktiven Editierbereich an, in welcher Zeile und Spalte sich der Cursor befindet. Die erste Zeile eines Textes hat Nummer Eins, die erste Spalte ganz links hat Nummer Null. Hinter der Spaltenanzeige gibt der Editor Meldungen aus, die sich auf den Puffer beziehen, z.B. bei einem Fehler während des Speicherns eines Puffers, daß die geschriebene Datei wieder aufgrund von Fehlern gelöscht wurde.

### 2.3.2 Icons

#### Das Standard-Icon

Der HiTex-Editor benutzt die Tooltypes der Icondatei eines Textes, um textspezifische Information abzuspeichern. Falls eine Textdatei beim Laden ein Icon besitzt, holt sich der Editor dieses und liest aus den Tooltypes die untenstehenden Werte. Beim Abspeichern werden die aktuellen Einstellungen der Werte ebenfalls in die Tooltypes geschrieben. Ihm unbekannte Tooltypes läßt der Editor dabei unverändert.

Hatte ein Textpuffer kein Icon, dann wird nur ein neues für ihn geschrieben, wenn die Preferences des Editors entsprechend eingestellt sind (siehe Kapitel 2.9 über die

Preferences). Die Form dieses neuen Icons können Sie vorgeben, indem Sie eine Icondatei namens ACOM:HiTex.default.info erzeugen. Beim Editorstart wird dieses Icon als Standard-Icon eingelesen. Beachten Sie dabei, daß Sie als Default-Tool des Icons den Dateinamen ACOM:Editor angeben, den Typ auf »Project« stellen und die Position nicht festlegen. Gibt es das obige Standard-Icon nicht als Datei, dann verwendet der Editor ein internes Standard-Icon.

### Icon-Tooltypes

Die dem Editor bekannten Tooltypes sind:

```
TABULATOR=zahl
LEFTMARGIN=zahl
RIGHTMARGIN=zahl
POSITION=zeile,spalte
WINDOW=x,y,breite,höhe,hidden
FLAGS=WT
MARK=zeile,spalte,name
```

Beim Laden werden die Werte entsprechend gesetzt. TABULATOR, LEFTMARGIN und RIGHTMARGIN werden u.a. bei den Preferences beschrieben. POSITION ist die Position des Cursors beim Speichern. WINDOW bezieht sich auf die Fenstereinstellung. Falls während des Befehls OPENNEWFILE oder beim Laden von Dateien während des Starts der WINDOW-Tooltype angegeben ist, öffnet sich das Fenster an dieser Position mit der entsprechenden Größe (u.U. ist der aktuelle Screen zu klein, und das Fenster wird entsprechend angepaßt), sofern die »hidden«-Angabe fehlt. FLAGS spiegelt die Einstellung der Flags Wordwrap und TabExpand wieder. MARK kann mehrfach vorkommen und ist eine gespeicherte Marke (siehe unten). Der Markenname gilt vom zweiten Komma an bis zum Zeilenende. Ein Beispiel:

```
MARK=145,16,StarteProgramm
```

Der Sachverhalt für den WINDOW-Tooltype ist beim Speichern etwas kompliziert. Gab es den Tooltype beim Laden und wird der Puffer gespeichert, bleibt der Tooltype unverändert. Wurde vor dem Speichern jedoch der Befehl SNAPSHOT ausgeführt, dann wird die Fensterposition und -größe gespeichert, die das Textpufferfenster zum Zeitpunkt von SNAPSHOT hatte, und es wird der Status angehängt, ob das Fenster gerade über den Befehl HIDEFILE geschlossen war (»hidden« fehlt, wenn das Fenster offen war). Nach dem Befehl UNSNAPSHOT wird gar kein WINDOW-Tooltype gespeichert, das Textfenster für den Puffer hat dann also keine Voreinstellung fürs Laden.

## 2.3.3 Marken

Jeder Textpuffer hat seine eigene Menge von Textmarken, die von denen der anderen Puffer unabhängig ist. Marken sind Positionsangaben innerhalb des Textes, die einen



symbolischen Namen haben. Man kann Marken setzen, löschen oder anspringen. Eine Marke sitzt auf einer Zeile und darin auf einer Spaltenposition und wird beim Einfügen oder Löschen von ein oder mehreren Zeichen oder Zeilen automatisch mitverschoben. Die Markennamen »S« und »E« sind für Blockstart und Blockende reserviert (siehe auch Kapitel über den Block).

Marken sind persistent, d.h., sie werden beim Schreiben eines Puffers in der zugehörigen Icondatei abgespeichert, so daß sie beim erneuten Laden des Textes zur Verfügung stehen. Es ist sinnvoll, z.B. die Prozedurnamen eines Quelltextes als Marken zu setzen, denn dann können sie einfach durch Klicken im Markenfenster angesprungen werden (siehe Kapitel 2.4.3 über das Markenfenster). Zusätzlich zu den Marken gibt es dann noch die Spezialmarke »OLDPOSITION« (dies ist die Tooltype-Angabe POSITION), die an der Stelle gesetzt wird, an der sich der Cursor beim letzten Schreiben der Datei befand. Auch der Block bleibt wegen der Marken »S« und »E« gespeichert und ist beim Einladen sofort markiert.

Während des Editierens kann es vorkommen, daß Marken ohne Rückmeldung gelöscht werden, z.B. wenn Sie mit dem Befehl DELETELINE die gesamte Zeile löschen, in der die Marke lag.

## 2.3.4 Texteingabe

### Overstrike

Der Editor befindet sich normalerweise immer im Einfügemodus. Wenn Sie eine Zeichentaste drücken, so fügt er das entsprechende Zeichen an der aktuellen Cursorposition ein und verschiebt alle rechts vom Cursor stehenden Zeichen und das Zeichen unter dem Cursor um eine Position nach rechts. Ist jedoch der Overstrike-Modus eingeschaltet, überschreibt das eingegegebene Zeichen das unter dem Cursor stehende, und der Editor verschiebt nicht. Beim normalen Einfügen meldet der Editor einen Fehler, wenn die Zeile zu lang wird. Bitte beachten Sie, daß der Mauszeiger verschwindet, wenn Sie Zeichen per Tastatur eingeben. Sobald Sie die Maus bewegen, erscheint er wieder. Das Ausschalten des Mauszeigers läßt sich in den Preferences verhindern (siehe dort).

Um den horizontal sichtbaren Bereich auszudehnen, schiebt der Editor den Text bei allen Funktionen automatisch nach rechts oder links (Scrolling). Sie können den Bereich auch ausdehnen, indem Sie das Fenster vergrößern.

### TabExpand

Tabulatorzeichen haben im Text eine besondere Bedeutung. Ist die Tabulatorexpansion eingeschaltet (siehe Preferences) und kommt bei der Textausgabe ein Tabulatorzeichen vor, so bewirkt dieses, daß die folgenden Zeichen erst ab der nächsten Tabulatorspalte auf dem Bildschirm erscheinen. Der Zwischenraum wird gelöscht, und der Cursor überspringt diesen Raum bei jeder Funktion. Dies können Sie benutzen, um bei Quelltexten Einrückungen mit so wenig Zeichen wie möglich durchzuführen. Die Breite des

Zwischenraumes ist von dem aktuellen Tabulatorabstand abhängig, und bei der Standardkonfiguration fügt ein Druck auf die Tabulator-Taste ein Tabulatorzeichen an der Cursorposition ein, so daß der nachfolgende Text der Zeile auf die nächste Tabulatorspalte springt. Ferner können am Zeilenanfang automatisch Tabulatorzeichen ergänzt werden, siehe ≫AutoSOLTab≪ bei Preferences. Haben Sie die Tabulatorexpansion abgeschaltet, dann expandiert der Editor keine Tabulatorzeichen im Text, sondern stellt sie direkt als Zeichen des eingeschalteten Fonts dar.

### Wordwrap

Für die Texteingabe existiert noch ein Modus, in dem Fließtext eingegeben werden kann: Der Wordwrap-Modus. Er wird über den Befehl SETPREFS ein- oder ausgeschaltet und bewirkt beim Einfügen oder Löschen von einzelnen Zeichen oder beim Befehl FORMATPARAGRAPH, daß der Absatz, in dem sich der Cursor befindet, von der Cursorposition an bis zum Absatzende so formatiert wird, daß die Worte genau zwischen dem linken und rechten Rand erscheinen. Dabei ist das Absatzende durch eine Leerzeile und ein Wort als Einheit von Zeichen definiert, die weder Leer- noch Tabulatorzeichen sind. Der Cursor steht nach all dem Verschieben immer noch an der Stelle des Wortes, an dem gerade geschrieben wird, auch wenn es auf die Folgezeile gerutscht ist.

Brauchen Sie ein Leerzeichen, das nicht durch Wordwrap getrennt werden soll, dann fügen Sie einfach ein sogenanntes hartes Leerzeichen mit dem ASCII-Code $A0 z.B. über den Befehl ≫INSERTASCII $A0≪ ein. Dieses Zeichen wird wie ein Leerzeichen ausgegeben, aber wie ein echtes Zeichen mit Inhalt wie z.B. ≫a≪ o.ä. behandelt. Beachten Sie dabei aber, daß manche Compiler und Assembler dieses Zeichen nicht verarbeiten können. Z.B. beim Optimierenden Makro-Assembler ist es nur in Kommentaren erlaubt und führt in anderen Fällen zu Fehlern.

Durch den Wordwrap-Modus kann vermieden werden, daß Ihnen im Eifer des Schreibens der Cursor über den rechten Rand gerät und der Text unbündig wird. Dies ermöglicht z.B. das bequeme Schreiben von Briefen oder längeren Texten, denn Sie müssen sich nie um das Zeilenende kümmern, nur um das Ende eines Absatzes. Dieses Handbuch würde übrigens mit Hilfe des Wordwrap-Modus auf dem HiTex-Editor selbst erstellt.

## 2.3.5 Der Block und die Maus

### Der Block

Als Block wird ein markierter Textbereich verstanden, auf den sich Blockoperationen beziehen. Ein Block kann z.B. ausgeschnitten und an einer anderen Stelle wieder eingeklebt werden. Vielleicht soll aber auch der durch den Block markierte Ausschnitt in eine Datei geschrieben werden. Das Markieren eines Textbereiches wird als Blocklegen bezeichnet. Der gelegte Block ist immer optisch hervorgehoben und somit gut erkennbar.



Sie können den Block mit Hilfe der Maus (siehe folgende Absätze) oder durch Setzen der Marken ≫S≪ für Blockstart und ≫E≪ für Blockende legen. Beim Legen per Maus werden diese beiden Marken automatisch auf die entsprechenden Positionen gesetzt. Der Block existiert genau dann, wenn beide Marken im Puffer gesetzt sind. Ob der Blockstart im Text tatsächlich vor dem Blockende liegt, ist für die Operationen des Editors unerheblich.

### Einfachklick

Die Maus ist ein wichtiges Hilfsmittel des Editors. Wenn Sie ein Zeichen mit der Maus anklicken, wird der Cursor auf diesem positioniert. Damit können Sie auch zwischen Editierbereichen hin- und herschalten.

### Doppelklick

Falls Sie ein Zeichen doppelt anklicken, das keine öffnende oder schließende Klammer ist und zu einem Wort gehört, dann wird der Block um das Wort gelegt, auf dem der Cursor steht (siehe auch Befehl SELECTWORD). Falls das Zeichen aber eine öffnende bzw. schließende Klammer ist, sucht der Editor die dazu passende schließende bzw. öffnende Klammer und positioniert den Cursor auf sie (siehe auch Befehl MOVEMATCHBKT. Für die Klammersuche sind runde, eckige, geschweifte und spitze Klammern gültig:

```
( )   [ ]   { }   < >
```

Diese Funktion erleichtert Ihnen das Überprüfen von Verschachtelungen in Hochsprachen, z.B. bei Befehlsblöcken in C oder bei Ausdrücken wie (a*b*(c+a)*((c+d)-d+(a))). Falls das doppelt geklickte Zeichen weder ein Wortzeichen noch eine Klammer ist, bewirkt der Doppelklick nichts außer dem Vermerk, daß nun auch ein Dreifachklick möglich ist.

### Dreifachklick

Nach einem Doppelklick ist auch ein Dreifachklick möglich. Damit wird der Block wie beim Befehl SELECTLINE um die ganze angeklickte Zeile gelegt. Dreifachklick auf eine Klammer bewirkt wiederum nur die Klammersuche. Bei mehr als dreimaligem Anklicken wird wieder beim Einfachklick begonnen, d.h., es gibt keinen Vierfachklick o.ä. Der ganze Text kann mit dem Befehl SELECTALL markiert werden.

### Ziehen der Maus

Mit der Maus wird bekanntlich gezogen, indem man anklickt und bei gedrückter Maustaste den Mauszeiger verschiebt. An der Endposition wird die Taste wieder losgelassen. Das Ziehen einer Trennzeile verändert die Größe ihres zugehörigen Editierbereiches.

Mit der Maus kann auch ein beliebiger Textausschnitt als Block markiert werden. Dazu klicken Sie den Blockstart an und ziehen den Cursor auf das Blockende. Der Editor meldet währenddessen, daß er den Block legt, färbt ihn ein und führt ein automatisches Scrolling nach oben, unten, rechts oder links durch, falls Sie an den Rand des Editierbereiches geraten. Das Scrollen ist umso schneller, je weiter der Mauszeiger von dem Editierbereich entfernt ist. Nach dem Loslassen der Maustaste sind die Marken »S« und »E« gesetzt, und der Block ist markiert.

Mittels Ziehen des Scrollbars kann außerdem der Textausschnitt, der in dem entsprechenden Editierbereich dargestellt wird, ausgewählt werden.

### 2.3.6 Privater ARexx-Port

Neben dem Hauptport des Editorprozesses besitzt jeder Textpuffer noch einen privaten ARexx-Port, dessen Name mit »HITEX.« beginnt und mit einer eindeutigen Zahl zur Identifikation endet (z.B. »HITEX.3«). Soll sich ein ARexx-Programm speziell auf nur einen Puffer beziehen, dann kann dieser ARexx-Port dafür benutzt werden, die Ausführung vom vom Benutzer aktuell gewählten Puffer unabhängig zu machen. Siehe auch Kapitel 2.7.1.



## 2.4 Hilfsfenster und Requester

### 2.4.1 Gemeinsamkeiten und Unterschiede

Neben den Textpufferfenstern besitzt der Editor noch etliche Hilfsfenster und Requester. Diese besitzen sogenannte Gadgets zur Eingabe von Werten, darunter Checkboxen, Stringfelder, Listen u.ä. Sie öffnen sich i.a. so in der Nähe des Mauszeigers, daß eines der Gadgets unter dem Zeiger liegt, damit die Bedienung ohne viel Mausbewegung möglich ist und z.B. ein einfacher Mausklick dieses Gadget mit der vom Editor erwarteten Standardantwort auslösen kann. Ist ein Hilfsfenster oder Requester in der Größe verstellbar, dann öffnet es sich später mit genau der gleichen Größe.

Ein Hilfsfenster ist nicht-modal, d.h., es kann offenstehen und erlaubt es, andere Fenster zu aktivieren und zu benutzen, ohne das Hilfsfenster zu schließen. Die Gadgets eines Hilfsfensters besitzen oft Tastaturkürzel. Diese Kürzel sind im Text zum Gadget unterstrichen, und bei aktivem Hilfsfenster genügt das Drücken der entsprechenden Taste, um das Gadget auszulösen. Die verschiedenen Gadgettypen reagieren auf das Kürzel wie folgt:

- Buttons werden ausgelöst.
- Checkboxes werden zwischen »angewählt« und »nicht angewählt« hin- und hergeschaltet.
- Cycle-Gadgets werden weitergestellt. Wird aber gleichzeitig die Shift-Taste gedrückt, dann wird zurückgestellt (dies gilt auch beim Anklicken des Gadgets mit der Maus bei gedrückter Shift-Taste).
- Integergadgets und Stringgadgets werden aktiviert.

Ein Requester ist hingegen modal, d.h., er setzt für alle anderen vor ihm geöffneten Fenster und Requester einen Warte-Mauszeiger, blockiert sie und verlangt die Eingabe einer Antwort. Gadgets und Menüs der anderen Fenster sowie ARexx-Befehle können währenddessen nicht ausgelöst werden, und erst nach Beantwortung des Requester kann normal weitergearbeitet werden. Die Gadgets eines Requesters befinden sich in der untersten Zeile und haben aufwendigere Tastaturkürzel, damit sie nicht aus Versehen beim Schnelltippen verschwinden. <Links-Amiga>+<V> bzw. <Links-Amiga>+<B> löst das ganz linke bzw. rechte Gadget aus. Darüberhinaus kann mit <Links-Amiga>+<Zahlentaste> das Gadget ausgelöst werden, das in der Gadgetzeile an der entsprechenden Position steht, z.B. ist <LAmiga>+<V> dasselbe wie <LAmiga>+<1>, und das dritte Gadget von links würde durch <LAmiga>+<3> ausgelöst. Requester besitzen nie ein Close-Gadget, damit die Benutzerantwort immer eindeutig bleibt.

Sowohl Hilfsfenster als auch Requester besitzen noch eine Gemeinsamkeit bei String- und Integergadget: Mit der Tabulator-Taste kann zum nächsten String-/Integergadget weitergeschaltet werden, mit zusätzlichem Shift wird zurückgeschaltet. Da bei einigen Fenstern oder Requestern durch Drücken der Return-Taste in einem Stringgadget bereits eine Funktion ausgelöst werden kann und dies nicht immer gewünscht ist, weil man noch andere Gadgets per Tastatur bedienen möchte, kann aus jedem solchen Gadget per Help-Taste herausgekommen werden, ohne die Funktion auszulösen.

### 2.4.2 Das Pufferlistenfenster

Im Pufferlistenfenster ist die Liste aller existierenden Puffer enthalten. Da es ein Hilfsfenster ist, darf es neben den Textfenstern aufbleiben, so daß durch einfach Mausklick der aktuelle Puffer gewechselt werden kann. Einfaches Anklicken eines Puffernamens in der Liste bewirkt nämlich, daß der Puffer mit seinem Fenster nach vorne geholt und aktiviert wird. Sollte sein Fenster gerade nicht offen sein, weil der Befehl HIDEFILE verwendet wurde, wird das Fenster für ihn geöffnet. Jeder Listeneintrag enthält zuerst das Modifikationsflag des Puffers (Plus- oder Minuszeichen), ein Flag, ob das Fenster offen ist (Pluszeichen) oder per HIDEFILE geschlossen wurde (Minuszeichen) und den Puffernamen. Vor dem Namen steht ein Stern »*«, wenn es der aktuelle Puffer ist. Hinter dem Namen folgt die Größe des Puffers in Bytes, eingefaßt in runden Klammern. Da diese aus Performancegründen nicht ständig mitgeführt wird, muß sie für die Puffer bei Bedarf durch Anklicken des Update-Gadgets aktualisiert werden. Schließlich folgt der ARexx-Name des privaten Ports für diesen Puffer in geschweiften Klammern. Neben dem Update-Gadget gibt es noch ein Cancel-Gadget, das wie das Close-Gadget das Pufferlistenfenster einfach schließt.

Das Pufferlistenfenster öffnet sich beim Befehl OPENBUFFERLIST mit seiner alten Größe unter dem Mauszeiger. Sollte das Fenster aber bereits offen sein, dann wird es einfach nach vorne gebracht, steht also an seiner alten Position und nicht unbedingt beim Mauszeiger.

### 2.4.3 Das Markenfenster

Auch das Markenfenster ist ein Hilfsfenster. Es zeigt die gesetzten Marken des aktuellen Textpuffers an. Wird der Puffer gewechselt, ändert sich auch die Anzeige im Markenfenster. Einfaches Anklicken einer Marke bewirkt, daß die Marke angesprungen und das aktuelle Textfenster aktiviert wird. Mit dem Delete-Gadget kann eine Marke schnell und effektiv gelöscht werden. Dazu muß »Delete« angeklickt werden, woraufhin sich der Mauszeiger in ein Löschsymbol ändert. Danach klicken Sie die zu löschende Marke an. Haben Sie es sich anders überlegt und wollen doch nicht löschen, klicken Sie entweder das Delete-Gadget ein zweites Mal an (der Mauszeiger normalisiert sich dann wieder) oder aktivieren irgendein anderes Fenster. Sobald das Markenfenster deaktiviert wird, ist der Delete-Modus abgeschaltet und der Mauszeiger wieder normal. Neben dem Delete-Gadget gibt es noch ein Cancel-Gadget, das wie das Close-Gadget das Markenfenster einfach schließt.

Das Markenfenster öffnet sich beim Befehl OPENBOOKMARKS mit seiner alten Größe unter dem Mauszeiger. Sollte das Fenster aber bereits offen sein, dann wird es einfach nach vorne gebracht, steht also an seiner alten Position und nicht unbedingt beim Mauszeiger.

### 2.4.4 Der Filerequester

Alle Funktionen des Editors, die eine Dateinamen-Auswahl benötigen (wie z.B. OPENFILE oder SAVEPREFS), verwenden dazu den Filerequester. Dieser ist ein Requester



im obigen Sinne und blockiert die anderen Fenster. Es ist in den Preferences einstellbar, ob ein eigener Filerequester oder der System-Standard-Filerequester verwendet werden soll. Da der letztere stark von der Betriebssystemversion abhängt, verweise ich auf die entsprechende Beschreibung in der dortigen Dokumentation. Hier wird nur der eigene Requester beschrieben. Dieser Filerequester kann Verzeichnisse mit bis zu 65535 Namen bearbeiten. Ein solches Verzeichnis wäre unter Amiga-DOS 32 MegaByte groß. Der Filerequester hat die Eigenschaft, daß sie während des Einlesens der Namenliste beliebig zwischen Gadgets und Verzeichnissen umschalten können.

Sobald der Requester erscheint, können Sie den Dateinamen im **Filename-Gadget** editieren. Je nach Funktion kann sich dort schon ein Vorschlag des Editors befinden. Wenn Sie dieses Gadget durch Drücken der Return-Taste verlassen, wird der Requester beendet und die aufrufende Funktion ausgeführt. Falls der Dateiname leer ist, wird die jeweilige Funktion abgebrochen. Um das Beenden des Filerequesters zu verhindern, können Sie wie oben erwähnt das Gadget auch per Tabulator- oder Help-Taste verlassen.

Das **Directory-Gadget** enthält immer den vollständigen Pfadnamen des aktuellen Verzeichnisses. Durch Anklicken des Gadgets und Ändern des Namens wird auf das angegebene Verzeichnis geschaltet, das Filename-Gadget aktiviert und die Namenliste neu erstellt. Falls kein Verzeichnis mit dem Namen existiert, blinkt der Screen einmal, und das Gadget wird erneut aktiviert.

Das **Parent-Gadget** bewirkt, daß auf das hierarchisch höherstehende Verzeichnis geschaltet wird. Das Filename-Gadget wird aktiviert und die Namenliste neu erstellt. Falls es sich bereits um das Wurzelverzeichnis handelt, passiert nichts.

Das **Pattern-Gadget** bestimmt das Auswahlmuster und damit, welche Dateien in der Liste sichtbar sind. Ein kleiner Nachrichtentext über der Namenliste teilt mit, wieviele Dateien unsichtbar sind. Der Inhalt des Pattern-Gadgets ist auch in den Preferences verzeichnet und kann abgespeichert und damit für den nächsten Editorstart verfügbar gemacht werden (siehe Kapitel über die Preferences).

Die **Namenliste** enthält die sichtbaren Namen des Verzeichnisses, wobei die Verzeichnisnamen vor den Dateinamen (diese haben eine Größenangabe) stehen und jeder dieser beiden Bereich alphabetisch sortiert ist. Durch Anklicken können Sie auf das entsprechende Verzeichnis schalten, woraufhin die Namenliste neu erstellt wird. Einfaches Anklicken eines Dateinamens übernimmt diesen in das Filename-Gadget und aktiviert es. Bei einem Doppelklick wird zusätzlich der Requester beendet und die den Filerequester aufrufende Funktion ausgeführt. Der Doppelklick funktioniert nicht bei abspeichernden Funktionen wie z.B. SAVEFILEAS oder SAVEMACRO, damit nicht aus Versehen eine falsche Datei überschrieben wird. Bei Befehlen wie z.B. SETHOMEDIR, die ein Verzeichnis auswählen, sind die Dateinamen in der Namenliste nicht anklickbar.

In der **Driveliste** sind alle Volumes, alle mit `Assign` zugewiesenen Verzeichnisse sowie verwendbare Devices alphabetisch sortiert verzeichnet. Anklicken schaltet auf das entsprechende Verzeichnis und aktiviert das Filename-Gadget. Dadurch können Sie in der von Ihnen durch `Assign` gestalteten Umgebung schnell zum Zielverzeichnis gelangen.

Das **Ok-Gadget** beendet den Requester und startet die den Requester aufrufende Funktion mit dem aktuellen Verzeichnis und dem aktuellen Dateinamen. Das Cancel-Gadget erlaubt Ihnen, den Requester zu beenden und die den Requester aufrufende

Funktion abzubrechen. Das Gadget mit der Aufschrift »Set home« setzt das Heimatverzeichnis für neu erzeugte Puffer wie beim Befehl SETHOMEDIR auf das aktuelle Verzeichnis des Filerequesters.

Bitte beachten Sie, daß Sie den Aufbau der Namenliste zeitweilig anhalten können, indem Sie die rechte Maustaste einmal drücken und wieder loslassen. Mit der gleichen Aktion können Sie den Aufbau fortsetzen. Während des Anhaltens können Sie Namen, Devices, den Scrollbar u.ä. auswählen. Falls irgendein Amiga-DOS-Fehler beim Einlesen der Namenliste entsteht (z.B. das aktuelle Verzeichnis nicht mehr verfügbar ist, weil die Diskette schon herausgenommen wurde), meldet der Filerequester diesen Fehler mit seinem Amiga-DOS-Fehlercode in dem Nachrichtentext über der Namenliste und läßt die Namenliste leer.

Der Filerequester erscheint anfangs in der vollen Höhe des sichtbaren Screenbereiches und kann in der Größe geändert werden. Er öffnet sich wieder in der Größe, in der er verlassen wurde, und mit der Namenliste unter dem Mauszeiger. Eine Besonderheit gibt es noch: Wenn ein Workbench-Icon in den Filerequester gezogen wird, so wird das Verzeichnis des Icons und sein Dateiname in den Filerequester eingetragen.

### 2.4.5 Das Findfenster

Das Findfenster ist ein Hilfsfenster. Es wird ausführlich im Kapitel 2.6 über die Suchfunktion beschrieben.

### 2.4.6 Die Preferences-Requester

Die Preferences-Requester sind ausführlich im Kapitel 2.9 über die Preferences beschrieben.

### 2.4.7 Der Print-Requester

Dieser Requester erscheint, solange gedruckt wird. Er enthält nur ein einziges Gadget namens »Stop«, das den Druckvorgang abbricht.

### 2.4.8 Der Assemble-Requester

Hier sind die Einstellungen für den residenten Assembler für die Befehle OPENASSEMBLE und ASSEMBLE enthalten (siehe auch dort).

### 2.4.9 Der Modified-Project-Requester

Dieser Requester erscheint, wenn ein veränderter Puffer gelöscht, geschlossen oder überladen o.ä. werden soll. Er fragt, ob der Puffer vor dem Fortfahren gesichert werden soll, ob gleich fortgefahren werden soll oder ob die Funktion abzubrechen ist.



## 2.5 Tastatur-Funktionen und Menüs

Dieses Kapitel beschreibt kurz die Belegung der Tastatur und der Menüs, die in der Standardkonfiguration festgelegt ist. Bitte beachten Sie, daß Sie die Editor-Konfiguration vollständig selber definieren können, siehe Kapitel 2.10. Die Standardkonfiguration des Editors ist in der Datei HiTex.prefs enthalten, dort finden Sie auch die Zuordnung der Menüs zu den jeweiligen Befehlen. Die meisten Menüs haben die gleiche Namensbezeichnung wie der entsprechende Editor-Befehl.

Wie oben bereits erwähnt, werden normalerweise Zeichen eingefügt. Ein Tastendruck erzeugt das Zeichen, das der Taste durch die Tastaturbelegung zugeordnet ist. Es gibt jedoch Spezialtasten (wie z.B. <Return>) und Tastenkombinationen, die andere Auswirkungen haben. Im folgenden ist die jeweilige Wirkung unter der Standardkonfiguration des Editors beschrieben.

<Shift>+<Keypad-Null> schaltet den Overstrike-Modus ein oder aus (Befehl »SETPREFS OverStrike Toggle«). Haben Sie keinen Zahlenblock, sollten Sie die Funktion umbelegen.

Die Delete-Taste löscht das Zeichen unter dem Cursor und zieht alle rechts vom Cursor stehenden Zeichen eine Spalte nach links (Befehl DELETE). Mit <Shift>+<Delete> wird auch im Overstrike-Modus ein Leerzeichen an der aktuellen Cursorposition eingefügt (Befehl INSERTSPACE). Hier wird der Wordwrap-Modus beachtet.

Die Backspace-Taste löscht das Zeichen vor dem Cursor und zieht alle rechts vom Cursor stehenden Zeichen und das Zeichen unter dem Cursor eine Spalte nach links (Befehl BACKSPACE). Falls der Cursor am Anfang der Zeile steht, wird diese Zeile an die vorige angehängt. Würde dadurch eine Zeile mit mehr als 254 Zeichen entstehen, meldet der Editor einen Fehler und bricht die Funktion ab.

Die Return-Taste bewirkt das Teilen der aktuellen Zeile an der Cursorposition und das Setzen des Cursors auf den eingestellten linken Rand (Befehl NEWLINE). Es wird also eine weitere Zeile erzeugt. Würde dadurch eine Zeile mit mehr als 254 Zeichen entstehen, meldet der Editor einen Fehler und bricht die Funktion ab. Wenn Sie die Einstellung des linken Randes ändern, rückt der Editor den Cursor anders ein. Mit dem linken und rechten Rand können Sie umgehen wie auf einer Schreibmaschine, aber beachten Sie, daß der Editor bei eingeschalteter Tabulatorexpansion so viele Tabulatorzeichen (und anschließend Leerzeichen) einfügt, wie zum Erreichen des linken Randes notwendig sind. Bei ausgeschalteter Expansion werden nur Leerzeichen eingefügt.

Die Enter-Taste hat die gleiche Funktion wie die Return-Taste.

Die Help-Taste bewirkt das Teilen der aktuellen Zeile an der Cursorposition. Im Gegensatz zur Return-Taste stellt der Editor hier jedoch fest, wieviel Leer- und Tabulatorzeichen in der alten Zeile vor dem ersten Zeichen standen und setzt den Cursor auf den gleichen Abstand in der neuen Zeile (Befehl NEWLINEINDENT). Dies wird in blockorientierten Sprachen wie C, Modula-2 oder Cluster benötigt, um bequem Instruktionen einzurücken, und wird als Indent-Funktion bezeichnet. Sowohl bei der Help- als auch bei der Return-Taste stehen die Zeichen, die auf der alten Zeile hinter bzw. unter dem Cursor standen, auch in der neuen Zeile hinter bzw. unter dem Cursor.

Die Tabulator-Taste fügt bei eingeschalteter Tabulatorexpansion ein Tabulatorzeichen an der aktuellen Cursorposition ein. Bei ausgeschalteter Expansion oder in Verbindung mit der Control-Taste bewirkt sie das Anspringen der nächsten Tabulatorspalte. Mit <Shift>+<Tabulator> oder <Control>+<Shift>+<Tabulator> wird die vorige Tabulatorspalte angesprungen, so daß Sie sich schnell innerhalb einer Zeile bewegen können. Der Tabulaturabstand ist wählbar (siehe Preferences).

Die Umschalttasten Shift, Alternate, Control und Amiga alleine haben keine Funktion. Mit der rechten Amiga-Taste kürzen Sie die Menüs wie gewohnt ab (sogenannte Shortcuts), z.B. ruft <Amiga>+<V> dem Befehl PASTE auf.

Die Funktionstasten können Sie beliebig mit Makroaufrufen belegen, d.h., es wird jeweils ein ARexx-Skript ausgeführt.

Die Cursortasten funktionieren je nach Tastenkombination unterschiedlich:

- Alleine gedrückt ist es die normale Cursorbewegung nach oben, unten, links und rechts (mit automatischem Scrolling).
- Mit <Shift> geht der Cursor auf:

  **<Cursor-up>** das erste Zeichen der obersten, sichtbaren Zeile, die noch nicht zum scrollsensitiven Bereich gehört.

  **<Cursor-down>** das erste Zeichen der untersten, sichtbaren Zeile, die noch nicht zum scrollsensitiven Bereich gehört.

  **<Cursor-left>** das erste Zeichen der aktuellen Zeile.

  **<Cursor-right>** das letzte Zeichen der aktuellen Zeile.
- Mit <Alternate> wird bei <Cursor-left> und <Cursor-right> der Text unter dem Cursor verschoben, ohne die Cursorposition auf dem Bildschirm zu verlassen. <Alternate>+<Cursor-up> ruft den Befehl »PREVIOUS Page« auf und <Alternate>+<Cursor-up> den Befehl »NEXT Page«.
- In Verbindung mit der Control-Taste bewirken <Cursor-left> und <Cursor-right> die Befehle »PREVIOUS Word« und »NEXT Word«.
- In Verbindung mit <Control>+<Shift> bewirkt <Cursor-up> den Befehl »POSITION SOF« und <Cursor-down> den Befehl »POSITION EOF«.



## 2.6 Such-Funktion

### 2.6.1 Das Findfenster

Der Editor besitzt eine schnelle Suchfunktion, um Strings (d.h. Textteile) in Texten zu finden. Gefundene Strings können Sie durch alternative Ersetzstrings austauschen lassen. Gesucht wird grundsätzlich ab der aktuellen Cursorposition. Um Such- und Ersetzstrings eingeben zu können, gibt es das Findfenster. Es wird mit dem Befehl OPENFIND geöffnet und kann auch nach Ende des Suchens offengelassen werden. Beim Aufruf der Suchfunktion wird immer die aktuelle Einstellung des Findfensters verwendet, und die Suchoperationen bezieht sich auf das aktuelle Textfenster. Das Findfenster hat die folgenden Gadgets:

**Find-Gadget:** Es erlaubt die Eingabe des Suchstrings, welcher Wildcards (siehe unten) enthalten darf. Mit der Tabulator-Taste wird das Changetext-Gadget aktiviert, und mit der Help-Taste wird das Find-Gadget deaktiviert. Drücken Sie die Return-Taste im Find-Gadget, dann wird das Findfenster geschlossen und eine einfache Suchaktion gestartet. Das rechts vom Find-Gadget stehende Cycle-Gadget ermöglicht es, interaktiv zwischen fünf verschiedenen Suchstrings zu wählen. Der sichtbare Suchstring wird verwendet. Damit muß bei zwischenzeitlichem Wechsel der ursprüngliche Suchstring nicht erneut eingegeben werden. Wird das Gadget mit gedrückter Shift-Taste angeklickt, schaltet es rückwärts.

**Changetext-Gadget:** Hier wird der Ersetzstring eingegeben. Mit der Tabulator-Taste wird das Find-Gadget aktiviert, und mit der Help-Taste wird das Changetext-Gadget deaktiviert. Drücken Sie die Return-Taste im Changetext-Gadget, dann wird die Suchaktion gestartet, ohne daß das Findfenster geschlossen wird. Das rechts vom Changetext-Gadget stehende Cycle-Gadget ermöglicht es, interaktiv zwischen fünf verschiedenen Ersetzstrings zu wählen. Der sichtbare Ersetzstring wird verwendet. Wird das Gadget mit gedrückter Shift-Taste angeklickt, schaltet es rückwärts.

**IgnoreCase-Gadget:** Wenn die Checkbox markiert ist, dann wird bei der Suche nicht zwischen Groß- und Kleinschreibung unterschieden, d.h., der Suchstring »Motor« wird auch im Text »Otto-MOTOR« gefunden. Ist die Checkbox nicht markiert, werden Groß- und Kleinbuchstaben bei der Suche als verschieden angesehen.

**OnlyWholeWord-Gadget:** Wenn die Checkbox markiert ist, wird der Suchstring nur gefunden, wenn er als eigenständiges Wort steht. Dabei wird die Preference-Einstellung »Complex Words« berücksichtigt. Als Beispiel würde der Suchstring »ich« im Text »dich und mich« nicht gefunden, wenn diese Option eingeschaltet ist.

**ScanBackward-Gadget:** Wenn die Checkbox markiert ist, dann wird rückwärts gesucht. Keine Markierung heißt vorwärts suchen.

**StringFromBlock-Gadget:** Dieses Gadget übernimmt den aktuellen Block als Suchstring. Dazu muß dieser auf einer einzigen Zeile sein und maximal 127 Zeichen haben (sonst wird ein Fehler gemeldet). Wenn Sie in Ihren Quelltexten nach

einem Begriff suchen, der gerade in dem Text steht, können Sie somit das entsprechende Wort doppelt anklicken (und damit den Block um dieses Wort legen) und anschließend durch Anklicken dieses Gadgets den Suchstring entsprechend setzen.

**FindNext-Gadget:** Bei Anklicken dieses Gadgets sucht der Editor nach dem nächsten Vorkommen des Suchstrings in der gewählten Richtung und springt diese Position an.

**Change-Gadget:** Dieses Gadget ist nur eingeschaltet, wenn gerade ein Vorkommen des Suchstrings gefunden wurde. Mit seiner Anwahl wird das aktuelle Vorkommen durch den Ersetzstring ersetzt und nach dem nächsten Vorkommen gesucht.

**ChangeAll-Gadget:** Mit diesem Gadget werden unsichtbar alle Vorkommen des Suchstrings in der gewünschten Suchrichtung ohne Rückfrage durch den Ersetzstring ersetzt. Der Editor meldet dann, wie oft er den Suchstring ausgetauscht hat.

Das Findfenster ist so angelegt, daß ein Begriff sehr einfach gesucht werden kann. Öffnen Sie das Findfenster, geben Sie den Suchstring ein und drücken Sie danach die Return-Taste. Das Findfenster verschwindet, und die Suche beginnt. Falls das Findfenster offen bleiben soll, drücken Sie zuerst die Tabulator-Taste (das Changetext-Gadget wird aktiv) oder die Help-Taste (das Find-Gadget wird deaktiviert) und danach erst die Return-Taste. Es gibt außerdem den Befehl FIND, der es mit seinen Optionen erlaubt, den Suchtext erneut zu suchen, ohne das Findfenster zu benutzen (siehe ARexx-Befehle). Die Standardkonfiguration des Editors enthält solchermaßen realisierte Menübefehle für die Suche. Natürlich können Sie auch die Befehle auf Funktionstasten oder andere Tasten legen.

Möchten Sie mit Sichtkontrolle ersetzen, so starten Sie die Suche ohne Schließen des Findfensters und betätigen danach nur das FindNext- bzw. Change-Gadget, um Ersetzung zu überspringen bzw. auszulösen.

### 2.6.2 Wildcards

Innerhalb eines Suchstrings dürfen Sie sogenannte Wildcards benutzen, die Platzhalter für einen oder mehrere unbekannte Buchstaben sind:

? steht für genau einen beliebigen Buchstaben.

* steht für eine beliebig lange Zeichenkette, die auch die Länge 0 haben kann. Die Länge ist allerdings durch die maximale Zeilenlänge von 254 Zeichen beschränkt.

Es handelt es sich nicht um die gleichen Wildcards wie bei Amiga-DOS! Falls Sie die Zeichen »?« und »*« und den Backslash »\« selber suchen möchten, müssen Sie ihnen einen Backslash voranstellen. In einem Ersetzstring werden diese drei Zeichen wie normale Zeichen behandelt, sie haben dort also keine besondere Bedeutung.

Lautet der Suchstring z.b. »W?lle«, wird der Editor sowohl »Welle« als auch »Wolle« oder »Wille« finden. Die Wildcard »*« ist in der Benutzung etwas heikel, da der



Editor den String »Otto*motor« findet, wenn er »Otto« findet und irgendwo auf der gleichen Zeile nachfolgend (bzw. vorhergehend beim Suchen rückwärts) das Wort »motor« vorkommt. Deshalb die folgende Warnung:

**DIE WILDCARDS KÖNNEN ERHEBLICHEN SCHADEN ANRICHTEN, WENN SIE BEIM ERSETZEN MIT »CHANGE ALL« VERWANDT WERDEN! BENUTZEN SIE DAHER DAS INTERAKTIVE ERSETZEN.**

### 2.6.3 Besonderheiten

Die Suchfunktion bietet viele Möglichkeiten der Textbearbeitung, da Sie in den Stringkomponenten Control-Sequencen direkt eingegeben können. Vorausssetzung dafür ist allerdings, daß Sie im System-Preference-Programm IControl den Text-Gadget-Filter ausgeschaltet haben. Dann können Sie ein Tabulatorzeichen z.B. über den Suchstring <Control>+<I> oder ein sogenanntes hartes Leerzeichen (ASCII-Code $A0, erscheint als Leerzeichen, hat aber anderen Code) über <Alternate>+<Leertaste> suchen.

Ein Zeilenende (eine Zeile endet mit einem Linefeed, $0A) können Sie finden lassen, indem Sie beim Suchstring <Control>+<J> (entsprechend dem Linefeed-Code) eingeben. Sollten Sie dann ersetzen wollen, müssen Sie den Ersetzstring ebenfalls mit <Control>+<J> abschließen! **Andernfalls kann der Editor die seltsamsten Aktionen vollführen, da er auf ein Linefeed am Zeilenende angewiesen ist.**

Wollen Sie eine Leerzeile finden, geben Sie zweimal <Control>+<J> als Suchstring an. Der Editor ist hierbei aber nicht in der Lage zu ersetzen oder rückwärts zu suchen! Benutzen Sie zur Verarbeitung einer Leerzeile ein ARexx-Programm.

## 2.7 ARexx

ARexx ist die Skript-Programmiersprache, die seit Kickstart 2.0 jedem Amiga beiliegt. Mit ARexx ist es möglich, Programme von außen zu steuern, ohne daß der Benutzer einzugreifen braucht. Damit lassen sich auf den Grundfunktionen des Editors aufbauend beliebig komplexe Operationen realisieren. Da sich der Editor zudem frei konfigurieren läßt, kann eine solche Operation sogar einfach durch Tastendruck vom Editor aus aufgerufen werden.

Solange Sie den Editor nur normal benutzen möchten, brauchen Sie sich nicht um ARexx zu kümmern. Als erfahrener Benutzer werden Sie ARexx aber bald intensiv benutzen. In diesem Kapitel wird vorausgesetzt, daß Sie das ARexx-Kapitel im Amiga-Benutzerhandbuch gelesen haben und sich mit den Grundlagen und der Form von ARexx auskennen.

### 2.7.1 ARexx-Ports

#### Der Hauptport

Jedes Programm, das ARexx unterstützt, benötigt einen Port, an den ARexx Befehle schicken kann. Der HiTex-Editor besitzt einen Hauptport für diesen Zweck. Da mehrere Prozesse vom HiTex-Editor gleichzeitig laufen können, unterscheiden sich die Hauptportnamen anhand einer Nummer voneinander. Der Name des Hauptports setzt sich aus einem Präfix und einer Nummer zusammen. Das Präfix ist normalerweise `HITEX.AREXX.`, kann aber mittels der Startoption PORTNAME anders gewählt werden. Beim Start sorgt der Editor dafür, daß sein Hauptportname im System einzigartig bleibt, indem an das Präfix eine Nummer angehängt wird. Wenn z.B. bereits zwei HiTex-Editoren auf dem Rechner laufen, bekommt der dritte den Hauptportnamen `HITEX.AREXX.3`. Es ist wichtig zu beachten, daß nach Commodore-Empfehlung ein ARexx-Portname vollständig in Großbuchstaben verfaßt sein sollte und keine Leerzeichen enthalten darf, ansonsten wird der Port u.U. von ARexx nicht gefunden.

Der Name des ARexx-Hauptports erscheint im Copyright-Requester. Siehe ARexx-Befehl COPYRIGHT.

#### Der Privatport

Neben dem Hauptport des Editorprozesses besitzt jeder Textpuffer noch einen privaten ARexx-Port, dessen Name mit `HITEX.` beginnt und mit einer eindeutigen Zahl zur Identifikation endet (z.B. `HITEX.3`). Soll sich ein ARexx-Programm speziell auf nur einen Puffer beziehen, dann kann dieser ARexx-Port dafür benutzt werden, die Ausführung vom vom Benutzer aktuell gewählten Puffer unabhängig zu machen. Numerisch aufeinanderfolgende Portnamen können zu verschiedenen Editorprozessen gehören. Der Name eines Privatports findet sich im Pufferlistenfenster.



### 2.7.2 ARexx-Details

#### Befehlsausführung

Bevor ein Befehl in einem ARexx-Skript an den Editor gesandt wird, muß mit dem ARexx-eigenen Befehl »ADDRESS« der Hauptport des Editors oder ein Privatport eines Puffers angewählt werden. Wird ein ARexx-Befehl an einen Privatport geschickt, dann bezieht sich seine Ausführung auch auf den entsprechenden Puffer, z.B. bei OPENFILE wird nur in diesen Puffer geladen. Wird ein pufferbezogener Befehl an den Hauptport geschickt, arbeitet er auf dem vom Benutzer aktuell eingestellten Puffer. Da dieser während der Ausführung eines ARexx-Skripts durch den Benutzer gewechselt werden kann und die Befehle dann auf den neuen, aktuellen Puffer wirken, ist somit Vorsicht angebracht.

ARexx kennt für Skriptnamen ein Standardsuffix, so daß der Name eines Skripts in einem anderen Skript als Befehl benutzt werden kann. Der HiTex-Editor verwendet als Suffix für seine ARexx-Skripts das Suffix ».hit« (siehe auch ARexx-Handbuch).

Befehle, die eine Rückgabe an das ARexx-Skript haben, geben diese Rückgabe wie unter ARexx üblich in der Variablen RESULT zurück.

#### ARexx beim Editorstart

Wie im Kapitel 2.2 beschrieben wurde, versucht der Editor bei fehlender STARTUP-Startoption, ein ARexx-Skript namens HiTex.hit aus dem aktuellen Verzeichnis auszuführen. Gelingt dies nicht, probiert er es noch einmal mit dem Skript ACOM:HiTex.hit. Dies ermöglicht es Ihnen, weitergehende Konfigurationen einzustellen als mit den Preferences und der Setup-Datei (siehe auch Kapitel 2.10 über die Editorkonfiguration).

### 2.7.3 Probleme mit ARexx

Da ARexx die einzelnen Argumente eines Befehls durch Leerzeichen unterscheidet, ergeben sich einige Probleme. Dazu gehört z.B., daß bei Aufruf eines Makros aus der Ram-Disk ihr Volume-Name keine Leerzeichen enthalten darf (siehe auch Befehl EXECAREXXMACRO). Zudem müssen Strings, die als Argumente übergeben werden und Leerzeichen enthalten, doppelt in Hochkommata eingefaßt werden, sonst erscheinen die einzelnen Worte des Strings ungetrennt voneinander bei dem Programm, das den Befehl ausführen soll. Beispiel: Die beiden Befehle

```
RequestNotify 'Der Titel' 'Der innere Text'
RequestNotify "Der Titel" "Der innere Text"
```

erscheinen beim Programm als

```
RequestNotify Der Titel Der innere Text
```

Nun sind aber die beiden Strings nicht mehr voneinander unterscheidbar. Korrekt sind die Schreibweisen

```
RequestNotify '"Der Titel"' '"Der innere Text"'
'RequestNotify "Der Titel" "Der innere Text"'
```

ARexx entfernt die einzelnen Hochkommata, und die doppelten bleiben für das Programm sichtbar. Bei ihm erscheint der Befehl dann korrekt als

```
RequestNotify "Der Titel" "Der innere Text"
```

### 2.7.4 ARexx-Befehle zur Pufferverwaltung

Im folgenden sind alle ARexx-Befehle aufgeführt, die der HiTex-Editor versteht. Das Befehlswort ist immer das erste Wort, und die Argumente sind als Schablonen wie bei Amiga-DOS üblich angegeben. Die Befehle sind thematisch geordnet und erscheinen alle im Stichwortverzeichnis.

**OPENBUFFERLIST**

Öffnet das Pufferlistenfenster bzw. bringt es nach vorne, wenn es schon offen ist.

**NEWFILE HIDDEN/S, PORTNAME/F**

Erzeugt einen neuen Puffer und gibt den ARexx-Privatportnamen mit angehängter Nummer für die Eindeutigkeit zurück. PORTNAME kann den Portnamen fest spezifizieren, dabei wird keine Nummer angehängt. Als Verzeichnis bekommt der Puffer das Heimatverzeichnis des Editors. Falls HIDDEN angegeben ist, wird für den Puffer anfangs kein Fenster geöffnet.

**OPENFILE FILENAME, QUIET/S**

Lädt eine Datei in den Puffer. Wenn FILENAME nicht angegeben wird, erscheint der Filerequester. QUIET unterdrückt den Modified-Project-Requester. Falls in einem Requester Cancel gewählt wird, ist der Rückgabecode 5. Da der Editor keine Zeilen von mehr als 254 Zeichen Länge verarbeiten kann, teilt er solche Zeilen beim Laden auf und meldet dies. Sollte genau dieser Text bereits in einem Puffer sein, so fragt der Editor nach, ob er wirklich noch einmal geladen werden soll. Falls nicht, bricht er das Laden ab und schaltet den betroffenen Puffer in den Vordergrund.

**OPENNEWFILE FILENAME, HIDDEN/S, PORTNAME/K**

Erzeugt einen neuen Puffer, lädt gleich eine Datei hinein und gibt den ARexx-Privatportnamen mit angehängter Nummer für die Eindeutigkeit zurück. PORTNAME kann den Portnamen fest spezifizieren, dabei wird keine Nummer angehängt. Wenn FILENAME nicht angegeben wird, erscheint der Filerequester. Falls Cancel gewählt wird, ist der Rückgabecode 5. Falls HIDDEN angegeben ist, wird für den Puffer anfangs kein Fenster geöffnet.

**SAVEFILE NOBACKUP/S, NOICON/S**

Schreibt den Puffer in eine Datei. Wenn der aktuelle Puffername leer ist, wird er zuerst über den Filerequester abgefragt. Falls Cancel gewählt wird, ist der Rückgabecode 5. NOBACKUP unterdrückt auf jeden Fall die Erstellung eines Backup, NOICON unterdrückt entsprechend das Erzeugen eines Icons. Weitere



Information zum Backup einer Datei findet sich im Kapitel über die Preferences. Falls alles gut geht, setzt der Editor das Modifikationsflag des Puffers zurück, da der Text ja gesichert wurde.

**SAVEFILEAS FILENAME, NOBACKUP/S, NOICON/S**

Schreibt den Puffer unter dem FILENAMEn in eine Datei und ändert den Puffernamen auf den angegebenen Namen. Wenn es keinen FILENAMEn gibt, wird er zuerst über den Filerequester abgefragt. Falls Cancel gewählt wird, ist der Rückgabecode 5. NOBACKUP und NOICON arbeiten wie bei SAVEFILE.

**SAVEMODIFIEDFILES QUIET/S**

Schreibt alle veränderten Puffer in ihre Dateien. Wenn QUIET nicht angegeben ist, wird vor dem Schreiben für jeden veränderten Puffer gefragt, ob er tatsächlich geschrieben werden soll. Die Antwort kann »Yes«, »No«, »Save all«, »Save none« oder »Cancel« sein. Sollte »Save all« geantwortet werden, wird die Abfrage nicht mehr durchgeführt, sondern es werden alle Puffer geschrieben. »Save none« schreibt keinen der Puffer, und »Cancel« bricht die Funktion mit Rückgabecode 5 ab. »Save none« kommt beim Befehl QUIT ins Spiel, wenn der Editor beendet werden soll, ohne eine Datei abzuspeichern.

**SAVEFILEQUIT QUIET/S**

Schreibt den Puffer und führt dann QUIT aus. QUIET unterdrückt den Modified-Project-Requester für die anderen Puffer.

**SETFILEPATH FILENAME/F**

Ändert den Puffernamen. Wenn es keinen FILENAMEn gibt, wird er zuerst über den Filerequester abgefragt. Falls Cancel gewählt wird, ist der Rückgabecode 5.

**SETCURRENTDIR DIRNAME/F**

Setzt das Verzeichnis des Puffers. Wenn DIRNAME fehlt, erscheint der Filerequester. Falls Cancel gewählt wird, ist der Rückgabecode 5. Das Verzeichnis ist der Identifikatorname eines Puffers, der nicht im Fenstertitel sichtbar ist.

**SETHOMEDIR DIRNAME/F**

Setzt das Heimatverzeichnis des Editors. Wenn DIRNAME fehlt, erscheint der Filerequester. Falls Cancel gewählt wird, ist der Rückgabecode 5. Das Heimatverzeichnis bestimmt, mit welchem Verzeichnis ein neu erzeugter Puffer beginnt.

**SETSAVEDATE**

Setzt das Datum für einen abzuspeichernden Puffer auf die aktuelle Zeit. Wird der Puffer vor dem Abspeichern noch einmal verändert, hat die Datei das Datum der Abspeicherzeit, sonst hat sie das Datum des SETSAVEDATE-Zeitpunkts. Diese Funktion ist im Zusammenhang mit einem externen Compiler sinnvoll, da zwischendurch compiliert und der Text erst später abgespeichert werden kann, ohne daß das Dateiendatum erneut gesetzt wird und ein Make-Utility denkt, der Quelltext müsse erneut übersetzt werden.

**CLEARSAVEDATE**

Löscht das obige Datum. Die Datei des Puffers wird das Datum der Abspeicherzeit haben.

**SNAPSHOT ALL/S**

Legt für den Puffer fest, daß die Fensterposition und -größe beim Speichern als WINDOW-Tooltype in das Icon gespeichert wird. Mit ALL betrifft dies alle Textpuffer. Erst nach diesem Befehl können also neue Fensterwerte z.B. durch den Befehl SAVEFILE in das Icon des Puffers gelangen. Siehe auch Kapitel 2.3.2.

**UNSNAPSHOT ALL/S**

Legt für den Puffer fest, daß die Fensterposition und -größe beim Speichern nicht mehr als WINDOW-Tooltype in das Icon gespeichert wird. Dies bedeutet aber auch, daß überhaupt kein WINDOW-Tooltype geschrieben wird, der Puffer hat also nach dem Speichern keine Fensterwerte mehr. Mit ALL betrifft dies alle Textpuffer. Siehe auch Kapitel 2.3.2.

**MAKE FILENAME/F**

Ruft das interne Make-Utility auf. FILENAME ist der Name des zugrundeliegenden Makefiles. Wenn FILENAME nicht angegeben ist, wird makefile als Name verwendet. Siehe auch Kapitel 2.8 über Make.

**CLOSEFILE QUIET/S**

Schließt den Puffer mitsamt Fenster. QUIET unterdrückt den Modified-Project-Requester. Falls Cancel gewählt wird, ist der Rückgabecode 5.

**PRINTFILE**

Druckt den Pufferinhalt über das Printer-Device. Es erscheint der Print-Requester. Das Drucken kann durch Anklicken des Stop-Gadgets im Print-Requester abgebrochen werden.

**CLEARFILE QUIET/S**

Löscht den Pufferinhalt, läßt das Fenster aber offen. QUIET unterdrückt den Modified-Project-Requester. Falls Cancel gewählt wird, ist der Rückgabecode 5.

**HIDEFILE**

Schließt das Fenster eines Puffers, ohne den Puffer zu entfernen oder seinen Inhalt zu löschen. Der Puffer ist weiterhin vorhanden und kann durch Anklicken im Pufferlistefenster oder durch den Befehl ACTIVATEDOC erneut geöffnet werden.

**QUIT QUIET/S**

Beendet den Editor. Falls ein Puffer noch verändert ist, kommt ein Requester hoch, der mit QUIET unterdrückt werden kann. Dieser Requester ist derselbe wie bei SAVEMODIFIEDFILES. »Save none« beendet hier den Editor, ohne irgendeinen Puffer abzuspeichern, »Cancel« hingegen bricht die Funktion ab und läßt den Editor weiterlaufen.



### 2.7.5 ARexx-Befehle zur Blockverwaltung

**CUTBLK**

Schneidet den aktuellen Block aus dem Text und speichert ihn in das Clipboard. Andere Programme, die das Clipboard ebenfalls unterstützen, können den Text somit mit ihrer Paste-Funktion hereinholen. Falls sich der Cursor innerhalb des Blocks befand, kommt er auf den ehemaligen Blockanfang.

**COPYBLK**

Kopiert den aktuellen Block in das Clipboard, ohne ihn zu löschen.

**PASTE SETBLOCK/S**

Fügt den Inhalt des Clipboards an der aktuellen Cursorposition ein. Der Cursor befindet sich anschließend hinter dem eingefügten Text. Falls SETBLOCK angegeben ist, wird der Block sofort um das gerade eingefügte Stück gelegt, so daß es sich sogleich horizontal verschieben oder wieder löschen läßt.

**DELETEBLK QUIET/S**

Entfernt den aktuellen Block, ohne ihn im Clipboard zu speichern. QUIET unterdrückt die Nachfrage per Requester, ob der Block tatsächlich ohne Rückholmöglichkeit gelöscht werden soll. Falls Cancel gewählt wird, ist der Rückgabecode 5. Falls sich der Cursor innerhalb des Blocks befand, kommt er auf den ehemaligen Blockanfang.

**UNMARKBLK**

Löscht die Blockmarkierung, d.h. die Marken S und E.

**SHIFTLEFT /N**

Verschiebt die Zeilen des Blocks um die angegebene Spaltenanzahl nach links. Wenn die Anzahl nicht angegeben ist, wird über einen Requester nachgefragt. Falls Cancel gewählt wird, ist der Rückgabecode 5. Der Block gilt hier zeilenweise. Die Zeile mit Marke S gehört dazu, die Zeile mit Marke E aber nur, wenn E nicht auf Spalte 0 steht. Falls Zeilen nicht weit genug geschoben werden kann, weil Zeichen außer Leer- oder Tabulatorzeichen gelöscht würden, dann gibt es einen Fehler mit Rückgabecode 10, und die betroffenen Zeilen bleiben unverändert.

**SHIFTRIGHT /N**

Verschiebt die Zeilen des Blocks um die angegebene Spaltenanzahl nach rechts. Wenn die Anzahl nicht angegeben ist, wird über einen Requester nachgefragt. Falls Cancel gewählt wird, ist der Rückgabecode 5. Der Block gilt hier zeilenweise. Die Zeile mit Marke S gehört dazu, die Zeile mit Marke E aber nur, wenn E nicht auf Spalte 0 steht. Falls Zeilen zu lang würden, dann gibt es einen Fehler mit Rückgabecode 10, und die betroffenen Zeilen bleiben unverändert.

**OPENCLIP FILENAME/F**

Lädt eine Datei in das Clipboard. Wenn FILENAME nicht angegeben wird, erscheint der Filerequester. Falls Cancel gewählt wird, ist der Rückgabecode 5. Nach Laden der Datei kann deren Inhalt z.B. mit PASTE eingefügt werden.

**SAVEBLK FILENAME/F**

Schreibt den Block unter dem FILENAMEn in eine Datei. Wenn es keinen FILENAMEn gibt, wird er zuerst über den Filerequester abgefragt. Falls Cancel gewählt wird, ist der Rückgabecode 5. Im Gegensatz zu SAVEFILE wird hier kein Backup oder Icon erzeugt.

**UPPERCASE BLOCK**

Wandelt alle Zeichen im Block in Großbuchstaben. Siehe ARexx-Befehl UPPERCASE.

**LOWERCASE BLOCK**

Wandelt alle Zeichen im Block in Kleinbuchstaben. Siehe ARexx-Befehl LOWERCASE.

**SWAPCASE BLOCK**

Wandelt alle Zeichen im Block von Groß- und Kleinbuchstaben und umgekehrt. Siehe ARexx-Befehl SWAPCASE.

**CONV2SPACES WIDTH/N**

Ersetzt alle Tabulatorzeichen im Block durch soviel Leerzeichen, daß jeweils die nächste Tabulatorposition unter der angegebenen Schrittweite WIDTH erreicht wird. Falls WIDTH nicht angegeben ist, wird die aktuelle Tabulatoreinstellung verwendet. Der Block gilt hier zeilenweise. Die Zeile mit Marke S gehört dazu, die Zeile mit Marke E aber nur, wenn E nicht auf Spalte 0 steht. Diese Funktion benötigen Sie für die Übertragung von Texten auf andere Editoren, die Tabulatorcodes nicht expandieren können, oder evtl. beim Ausdrucken, wenn der Drucker eine falsche Tabulatoreinstellung hat.

**CONV2TABS WIDTH/N**

Ersetzt alle Leerzeichen im Block, die bis zu einer unter der angegebenen Schrittweite WIDTH gültigen Tabulatorposition reichen, durch ein Tabulatorzeichen. Falls WIDTH nicht angegeben ist, wird die aktuelle Tabulatoreinstellung verwendet. Der Block gilt hier zeilenweise. Die Zeile mit Marke S gehört dazu, die Zeile mit Marke E aber nur, wenn E nicht auf Spalte 0 steht. Diese Funktion entspricht einem Komprimieren eines Textes.

**SELECTWORD**

Markiert das aktuelle Wort als Block. Ob das Wort als komplex angesehen wird, ist durch die Preference-Einstellung bestimmt. Falls der Cursor unmittelbar hinter dem Wort steht, wird dieses Wort trotzdem markiert. Ansonsten geschieht nichts.

**SELECTLINE**

Markiert die aktuelle Zeile als Block.



**SELECTALL**

Markiert den gesamten Text des Puffers als Block.

**GETBLKINFO**

Gibt ON zurück, falls der Block gelegt ist, sonst OFF.

**GETBLK**

Gibt den Text im Block zurück. Dazu muß der Block auf einer einzigen Zeile liegen und maximal 127 Zeichen lang sein. Falls das nicht erfüllt ist, gibt es einen Fehler.

### 2.7.6 ARexx-Befehle zur Cursorbewegung

**GOTOLINE /N**

Springt auf eine Zeile mit der angegebenen Zeilennummer. Falls keine Zahl angegeben ist, wird sie über einen Requester abgefragt. Falls Cancel gewählt wird, ist der Rückgabecode 5.

**GOTOCOLUMN /N/A**

Springt auf eine Spalte. Auf den Befehl muß eine Spaltenzahl folgen.

**MOVE LINE/N/A, COLUMN/N/A**

Kombination von GOTOLINE und GOTOCOLUMN.

**GOTOLEFTMARGIN**

Springt auf den linken Rand. Der linke Rand kann durch SETPREFS eingestellt werden.

**GOTORIGHTMARGIN**

Springt auf den rechten Rand. Der rechte Rand kann durch SETPREFS eingestellt werden.

**CURSOR UP/S, DOWN/S, LEFT/S, RIGHT/S**

Bewegt den Cursor eine Zeile hoch/runter oder eine Spalte nach links/rechts. Falls dieser Befehl durch Tastendruck ausgelöst wurde, ist es bei hoher Tastenwiederholgeschwindigkeit möglich, daß um mehr als eine Zeile/Spalte bewegt wird. Falls bei UP bzw. DOWN bereits der Textanfang bzw. das Textende erreicht wurde, ist der Rückgabecode 5.

**SCROLL UP/S, DOWN/S, LEFT/S, RIGHT/S**

Bewegt den Cursor eine Zeile hoch/runter oder eine Spalte nach links/rechts, verschiebt aber den Text unter dem Cursor und läßt den Cursor auf derselben Bildschirmposition. Falls dieser Befehl durch Tastendruck ausgelöst wurde, ist es bei hoher Tastenwiederholgeschwindigkeit möglich, daß um mehr als eine Zeile/Spalte bewegt wird.

`POSITION SOF/S, EOF/S, SOV/S, EOV/S, SOL/S, EOL/S, SOW/S, EOW/S`

Bewegt den Cursor auf spezielle Position:

- SOF ist Start-of-File, EOF ist End-of-File.
- SOV ist Start-of-View, EOV ist End-of-View.
- SOL ist Start-of-Line, EOL ist End-of-Line.
- SOW ist Start-of-Word, EOW ist End-of-Word.

Dabei ist SOF der Textanfang, EOF das Textende (eigentlich das erste Zeichen der letzten Zeile), SOV die oberste sichtbare Zeile, die noch nicht zum scrollsensitiven Bereich gehört, EOV entsprechend die unterste, SOL der Zeilenanfang, EOL das Zeilenende, SOW der Anfang des Wortes und EOW das Ende des Wortes. Die Positionen können auch kombiniert werden, z.B. bringt der Befehl »POSITION SOV SOL« den Cursor an den Anfang der obersten sichtbaren Zeile.

`NEXT WORD/S, PAGE/S`

Bewegt den Cursor zum nächsten Wort bzw. zur nächsten Seite. Zum nächsten Wort heißt ans Ende des aktuellen Wortes bzw. des folgenden, wenn der Cursor nicht auf einem Wort steht. Beim seitenweisen Blättern wird die unterste sichtbare Zeile der alten Seite zur obersten der neuen. Der Cursor bleibt dabei auf seiner visuellen Position innerhalb des Editierbereiches. Zum Springen auf den Anfang des nächsten Wortes siehe MOVENEXTWORD. An das Ende des Wortes zu gehen ist besonders für Fließtext im Wordwrap-Modus zu empfehlen.

`PREVIOUS WORD/S, PAGE/S`

Bewegt den Cursor zum vorigen Wort bzw. zur vorigen Seite. Zum vorigen Wort heißt auf den Anfang des aktuellen Wortes bzw. des davorstehenden, wenn der Cursor nicht auf einem Wort steht. Beim seitenweisen Blättern wird die oberste sichtbare Zeile der alten Seite zur untersten der neuen. Der Cursor bleibt dabei auf seiner visuellen Position innerhalb des Editierbereiches.

`MOVEDOWN LINES/N`

Bewegt den Cursor um die angegebene Zeilenanzahl nach unten. Wenn keine Zeilenanzahl angegeben ist, wird eine Zeile nach unten bewegt.

`MOVEUP LINES/N`

Bewegt den Cursor um die angegebene Zeilenanzahl nach oben. Wenn keine Zeilenanzahl angegeben ist, wird eine Zeile nach oben bewegt.

`MOVELEFT CHARS/N`

Bewegt den Cursor um die angegebene Zeichenanzahl nach links. Wenn keine Zeichenanzahl angegeben ist, wird um ein Zeichen nach links bewegt.



`MOVERIGHT CHARS/N`

Bewegt den Cursor um die angegebene Zeichenanzahl nach rechts. Wenn keine Zeichenanzahl angegeben ist, wird um ein Zeichen nach rechts bewegt.

`MOVEUPSCR`

Blättert eine Seite hoch wie »PREVIOUS PAGE«.

`MOVEDOWNSCR`

Blättert eine Seite runter wie »NEXT PAGE«.

`MOVESOF`

Bewegt den Cursor auf den Textanfang (Start-of-File).

`MOVEEOF`

Bewegt den Cursor auf das Textende (End-of-File).

`MOVESOL`

Bewegt den Cursor auf den Zeilenanfang (Start-of-Line).

`MOVEEOL`

Bewegt den Cursor auf das Zeilenende (End-of-Line).

`MOVENEXTWORD`

Geht an den Anfang des nächsten Wortes. »NEXT WORD« hingegen geht an das Ende des nächsten Wortes.

`MOVEPREVWORD`

Geht an den Anfang des vorigen Wortes wie »PREVIOUS WORD«.

`MOVENEXTTABSTOP`

Bewegt den Cursor auf die nächste Tabulatorposition. Die Tabulatorpositionen sind durch die Einstellung des Tabulatorabstandes per SETPREFS bestimmt.

`MOVEPREVTABSTOP`

Bewegt den Cursor auf die vorige Tabulatorposition. Die Tabulatorpositionen sind durch die Einstellung des Tabulatorabstandes per SETPREFS bestimmt.

### 2.7.7 ARexx-Befehle zur Markenverwaltung

`OPENBOOKMARKS`

Öffnet das Markenfenster bzw. bringt es nach vorne, wenn es schon offen ist.

**SETBOOKMARK NAME/F**

Setzt eine Marke mit dem NAMEn an die aktuelle Cursorposition. Der Name ist ein String. Falls er nicht angegeben wird, kommt ein Requester zur Namenabfrage, der als Vorschlag den aktuellen Block enthält (sofern er auf einer Zeile ist und maximal 127 Zeichen hat). Falls Cancel gewählt wird oder der Markenname oder NAME leer ist, ist der Rückgabecode 5. Führende Leer- und Tabulatorzeichen im Markennamen werden ignoriert. Die Markennamen »S« und »E« sind für Blockstart und -ende reserviert, setzt man sie, wird dadurch automatisch der Block gelegt.

**CLEARBOOKMARK NAME/F**

Löscht eine Marke mit dem NAMEn. Falls der Name nicht angegeben wird, kommt ein Requester mit dem aktuellen Block als Vorschlag (sofern er auf einer Zeile ist und maximal 127 Zeichen hat). Falls Cancel gewählt wird oder der Markenname oder NAME leer ist, ist der Rückgabecode 5. Führende Leer- und Tabulatorzeichen im Markennamen werden ignoriert.

**GOTOBOOKMARK NAME/F**

Springt eine Marke an. Falls kein NAME angegeben, kommt ein Requester mit dem aktuellen Block als Vorschlag (sofern er auf einer Zeile ist und maximal 127 Zeichen hat). Falls Cancel gewählt wird oder der Markenname oder NAME leer ist, ist der Rückgabecode 5. Falls der Name mit dem Hashzeichen »#« beginnt, muß eine Dezimalzahl folgen, die als direkte Zeilennummer verstanden wird; diese Zeile wird dann angesprungen. Führende Leer- und Tabulatorzeichen im Markennamen werden ignoriert.

**MOVEBOOKMARK NAME/F**

Synonym für GOTOBOOKMARK aus Kompatibilitätsgründen.

### 2.7.8 ARexx-Befehle zur Fensterverwaltung

**ACTIVATEDOC**

Bringt das Fenster des mit diesem ARexx-Befehl angesprochenen Puffers in den Vordergrund und aktiviert es.

**ACTIVATELASTDOC**

Aktiviert das Fenster des aktuellen Puffers, bringt es in den Vordergrund und bringt auch den Screen des Editors in den Vordergrund. Falls es keinen aktuellen Puffer gibt, wird das letzte Editorfenster aktiviert.

**ACTIVATENEXTDOC**



Aktiviert den im Pufferlistenfenster auf den aktuellen Puffer folgenden Puffer und bringt sein Fenster und den Editor-Screen nach vorne. Nach dem letzten Puffer wird wieder der erste aktiviert.

**ACTIVATEPREVDOC**

Aktiviert den im Pufferlistenfenster vor dem aktuellen Puffer stehenden Puffer und bringt sein Fenster und den Editor-Screen nach vorne. Nach dem ersten Puffer wird wieder der letzte aktiviert.

**ACTIVATEWINDOW**

Aktiviert das letzte aktive Editorfenster. Dies kann auch ein Hilfsfenster sein.

**SPLITVIEW**

Teilt den aktuellen Editierbereich in zwei Hälften auf. Es entstehen zwei Editierbereiche mit gleicher Positionierung im Text. Der obere wird zum aktuellen Editierbereich. Falls im Fenster kein Platz für einen zusätzlichen Editierbereich ist, passiert nichts.

**DELETEVIEW**

Löscht den aktuellen Editierbereich und schaltet auf den darunter liegenden bzw. den darüber, wenn der unterste gelöscht wird. Falls das Fenster nur einen Editierbereich hat, passiert nichts.

**NEXTVIEW**

Aktiviert den unter dem aktuellen Editierbereich stehenden Editierbereich. Gibt es dort keinen weiteren, wird der oberste Editierbereich aktiviert.

**PREVVIEW**

Aktiviert den über dem aktuellen Editierbereich stehenden Editierbereich. Gibt es dort keinen weiteren, wird der unterste Editierbereich aktiviert.

**GROWVIEW**

Vergrößert den aktuellen Editierbereich um eine Textzeile, falls das möglich ist.

**SHRINKVIEW**

Verkleinert den aktuellen Editierbereich um eine Textzeile, falls das möglich ist.

**BEEPSCREEN**

Läßt den Screen, auf dem der Editor läuft, kurz aufblinken.

**CHANGEWINDOW NAME/K, XPOS/N/A, YPOS/N/A, WIDTH/N/A, HEIGHT/N/A**

Bewegt das aktuelle Fenster auf die Position (XPOS,YPOS) und skaliert es auf die Größe (WIDTH,HEIGHT). Wenn einer der Werte gleich `-1` ist, bleibt er unverändert (Achtung: ARexx versteht die `-1` nur, wenn sie als `'-1'` angegeben ist). Da Fenster keine Teile außerhalb der Screengrenzen haben können, ist es möglich, daß das Ergebnis nicht ganz mit den gewünschten Werten übereinstimmt, denn der Editor verschiebt das Fenster bzw. skaliert es so, daß es auf den Screen paßt. NAME kann »FIND«, »BUFFERLIST« oder »BOOKMARKS«

sein und bestimmt dann, daß dieses Fenster in seiner Größe und Position geändert wird. Falls das Fenster in dem Moment nicht offen ist, wird es sich beim nächsten Mal mit diesen Werten öffnen (siehe auch SAVEWINDOWINFO). Wenn NAME fehlt, wird das aktuelle Fenster betroffen.

**SAVEWINDOWINFO NAME/F**

Schreibt ein ARexx-Programm in die Datei NAME, das Information darüber enthält, wie beim nächsten Startup die Hilfsfenster zu öffnen sind. Falls NAME nicht angegeben ist, wird er über einen Requester abgefragt. Beispiel:

```
/* Startup WindowInfo */
CHANGEWINDOW FIND 10 10 '-1' '-1'
OPENFIND
CHANGEWINDOW NAME BOOKMARKS 20 100 200 400
OPENBOOKMARKS
```

Diese Datei kann z.B. beim Shell-Start als gleich abzuarbeitendes ARexx-Programm angegeben werden, so daß der Editor die Hilfsfenster genau dort öffnet, wo sie beim letzten Aufruf von SAVEWINDOWINFO standen. Eine andere Möglichkeit ist es, die Datei als Standard-Startup-Makro HiTex.hit abzuspeichern.

**WINDOW2FRONT**

Bringt das Textfenster in den Vordergrund und aktiviert es.

**WINDOW2BACK**

Bringt das Textfenster in den Hintergrund.

**SCREEN2FRONT**

Bringt den Screen, auf dem der Editor läuft, in den Vordergrund.

**SCREEN2BACK**

Bringt den Screen, auf dem der Editor läuft, in den Hintergrund.

**WB2FRONT**

Bringt den Workbench-Screen in den Vordergrund.

### 2.7.9 ARexx-Befehle für die Preferences

**OPENPREFWINDOW**

Öffnet den Preference-Requester. Alle anderen Fenster werden blockiert.



**OPENSCREENPREFWINDOW**

Öffnet den Screen-Preference-Requester. Alle anderen Fenster werden blockiert.

**OPENPREFS FILENAME/F**

Lädt eine Preferencedatei. Wenn FILENAME nicht angegeben wird, erscheint der Filerequester. Falls Cancel gewählt wird oder der Dateiname leer ist, ist der Rückgabecode 5. FILENAME sollte die komplette Pfadangabe enthalten.

**SAVEPREFS FILENAME/F**

Speichert eine Preferencedatei. Wenn FILENAME nicht angegeben wird, erscheint der Filerequester. Falls Cancel gewählt wird oder der Dateiname leer ist, ist der Rückgabecode 5. FILENAME sollte die komplette Pfadangabe enthalten.

**SETPREFS PREFS/A, SETTING/F**

Stellt einen Preferencewert PREFS auf die Einstellung SETTING. Siehe auch Kapitel über die Preferences. Als PREFS und SETTING sind die folgenden Möglichkeiten vorhanden:

```
ASLRequest [ ON | OFF | TOGGLE ]
AutoSOLTabs [ ON | OFF | TOGGLE ]
BackupMode [ OFF | T | BAK ]
ComplexWords [ ON | OFF | TOGGLE ]
ConsoleName <Consolename>
DefaultName <Defaultname>
FilePattern <Filepattern>
HideMouse [ ON | OFF | TOGGLE ]
LeftMargin <LinkerRand>
OpenSize <Width> <Height>
Overstrike [ ON | OFF | TOGGLE ]
Refresh [ SIMPLE | SMART ]
RightMargin <RechterRand>
SaveIcons [ ON | OFF | TOGGLE ]
TabExpand [ ON | OFF | TOGGLE ]
TabWidth <Tabulatorabstand>
WordWrap [ ON | OFF | TOGGLE ]
WrapAround [ ON | OFF | TOGGLE ]
```

**GETPREFS PREFS/A**

Gibt den aktuellen Preferencewert für PREFS zurück. PREFS kann die gleichen Werte wie bei SETPREFS annehmen.

**FETCHOPENSIZE**

Holt die Größe des aktuellen Textfensters und setzt die »OpenSize«-Einstellung der Preferences auf diesen Wert. Falls es kein aktuelles Textfenster gibt, bleibt »OpenSize« unverändert.

### 2.7.10 ARexx-Steuerbefehle und Makros

**EXECAREXXSTRING SYNC/S, ASYNC/S, MAIN/S, CONSOLE/S, MACRO/F**

Führt den String MACRO als ARexx-Programm aus. Falls MACRO nicht angegeben ist, kommt ein Requester hoch und fragt nach dem auszuführenden String. Falls Cancel gewählt wird oder der Name leer ist, ist der Rückgabecode 5. Mit SYNC kehrt der Befehl erst nach vollständiger Ausführung des Makros zurück, mit ASYNC können die Befehle beliebig eingestreut eintreffen. Sind beide nicht angegeben, wird synchron gearbeitet. MAIN bestimmt, ob der Default-Host für die Ausführung der ARexx-Hauptport oder der lokale Port des aktuellen Puffers ist (siehe auch ARexx-Handbuch). Mit CONSOLE wird für die Dauer der Ausführung ein Consolefenster geöffnet, dessen Dateiname durch den Console-Namen in den Preferences bestimmt ist (siehe auch Kapitel 2.9). Der eingegebene Befehl wird von Aufruf zu Aufruf gemerkt und als Vorschlag beim Öffnen des Requesters wie beim Workbench-Befehl »Execute command« angegeben.

**RXS SYNC/S, ASYNC/S, MAIN/S, CONSOLE/S, MACRO/F**

Synonym für EXECAREXXSTRING.

**EXECAREXXMACRO SYNC/S, ASYNC/S, MAIN/S, CONSOLE/S, MACRO/F**

Führt ein ARexx-Programm namens MACRO aus. Falls MACRO nicht angegeben ist, erscheint der Filerequester und fragt nach dem auszuführenden Programm. Falls Cancel gewählt wird oder der Dateiname leer ist, ist der Rückgabecode 5. Mit SYNC kehrt der Befehl erst nach vollständiger Ausführung des Programmes zurück, mit ASYNC können die Befehle beliebig eingestreut eintreffen. Sind beide nicht angegeben, wird synchron gearbeitet. MAIN bestimmt, ob der Default-Host für die Ausführung der ARexx-Hauptport oder der lokale Port des aktuellen Puffers ist (siehe auch ARexx-Handbuch). Mit CONSOLE wird für die Dauer der Ausführung ein Consolefenster geöffnet, dessen Dateiname durch den Console-Namen in den Preferences bestimmt ist (siehe auch Kapitel 2.9). ACHTUNG: Der komplette Pfadname des Programmes einschließlich Volume-Namen darf keine Leerzeichen enthalten, sonst findet ARexx das Programm nicht und meldet Fehler 1. Um z.B. ARexx-Programme aus der Ram-Disk zu starten, sollte Ihr Volumename gegebenenfalls so umbenannt werden, daß er keine Leerzeichen enthält.

**RX SYNC/S, ASYNC/S, MAIN/S, CONSOLE/S, MACRO/F**

Synonym für EXECAREXXMACRO.

**RECORDMACRO**

Startet die Makroaufnahme. Alle Menübefehle, Tastenbefehl und ARexx-Befehle (außer RECORDMACRO, ENDMACRO, PLAYMACRO und SAVEMACRO) werden ab jetzt aufgezeichnet. Für die Dauer der Makroaufnahme ändert sich der Mauszeiger in ein Kassettensymbol.

**ENDMACRO**

Beendet die Makroaufnahme. Der aufgenommene Makro kann nun per PLAYMACRO abgespielt werden. Falls keine Makroaufnahme läuft, passiert nichts.



**PLAYMACRO COUNT/N**

Führt den aufgenommenen Makro aus. Wenn COUNT fehlt, wird die Anzahl der Makroausführungen per Requester abgefragt. Wenn COUNT Null ist, wird der Makro immer wieder ausgeführt, bis ein Fehler auftritt. Sonst wird der Makro so oft ausgeführt, wie COUNT angibt.

**SAVEMACRO FILENAME/F**

Speichert den aufgezeichneten Makro als ARexx-Programm in eine Datei. Wenn FILENAME nicht angegeben wird, erscheint der Filerequester. Falls Cancel gewählt wird oder der Dateiname leer ist, ist der Rückgabecode 5. FILENAME sollte die komplette Pfadangabe enthalten.

**DEFINEKEY KEY/A, SHIFT/S, ALT/S, CONTROL/S, LAMIGA/S, COMMAND/A/F**

Überlädt eine Taste dynamisch wie auch bei der Setup-Datei möglich. KEY muß ein ASCII-Zeichen oder ein mnemonischer Name wie in der Setup-Datei sein (RETURN etc.). Es sind dieselben Qualifier wie in der Setup-Datei erlaubt. Beispiel: Auf die F1-Taste hin soll ein ARexx-Makro ausgeführt werden:

```
DefineKey F1 'RX RAM:ARexxMakro'
```

**REQUESTNOTIFY TITLE/A, PROMPT/A**

Öffnet einen Requester mit »Ok«-Gadget, TITLE als Titelstring und PROMPT als inneren Text.

**REQUESTBOOL TITLE/A, PROMPT/A**

Öffnet einen Requester mit »Ok«- und »Cancel«-Gadget, TITLE als Titelstring und PROMPT als inneren Text und liefert YES oder NO an ARexx zurück.

**REQUESTNUM TITLE/A**

Öffnet einen Requester zur Zahleneingabe. Die Zahl kommt als String an ARexx zurück. Falls Cancel gedrückt wird, ist der Rückgabecode 5.

**REQUESTSTR TITLE/K, DEFAULT/F**

Öffnet einen Requester zur Stringeingabe. TITLE kann den Titel des Requesters angeben, DEFAULT den Initialstring. Das Ergebnis wird als String an ARexx zurückgegeben. Falls Cancel gedrückt wird, ist der Rückgabecode 5.

**REQUESTFILE TITLE/K, PATTERN/K, PATH/F**

Öffnet den Filerequester mit dem Titel TITLE, dem Filepattern PATTERN und dem Verzeichnis PATH. PATH muß existieren, wenn es angegeben ist. Der vollständige Dateiname wird als String an ARexx zurückgegeben. Falls Cancel gedrückt wird, ist der Rückgabecode 5. Falls die Datei noch nicht auf Platte existiert, kann der Name falsch sein, sofern der Dateiname noch Verzeichnisangaben enthält.

**WRITECHARCODE**

Schreibt den Zeichencode des Zeichens unter dem Cursor als Dezimal- und Hexadezimalzahl auf die Statuszeile.

`WRITESTRING STRING/F/A`

Schreibt den STRING in die Statuszeile.

`SETPRIORITY /N/A`

Setzt die Task-Priorität des Editors auf den angegebenen Wert. **Vorsicht: Zu hohe Werte können üble Dinge für das System bedeuten!**

`GETPRIORITY`

Holt die Task-Priorität des Editors.

### 2.7.11 ARexx-Befehle zum Suchen/Ersetzen

`OPENFIND`

Öffnet das Suchfenster und aktiviert das Find-Gadget.

`FIND FORWARD/S, BACKWARD/S, CASEON/S, CASEOFF/S, WORDON/S, WORDOFF/S, FROMBLOCK/S, UNTIL/K, TEXT/F`

Sucht nach dem angegebenen String. Falls der Suchstring fehlt, wird der aktuell eingestellte genommen. Wenn die Schalter nicht angegeben werden, verwendet der Editor die aktuellen Einstellungen. FROMBLOCK holt den Suchstring aus dem aktuellen Block. Nach UNTIL muß ein Markenname folgen. Von der aktuellen Position aus wird gesucht, bis die Marke erreicht ist. Wenn die Marke vor der aktuellen Position liegt und vorwärts gesucht wird, findet der Editor nichts und bricht sofort ab. Nach der Ausführung sind die obigen Einstellungen in das Findfenster übernommen. Anmerkung: Das UNTIL-Feature ist vorerst nicht implementiert. Wenn der String nicht gefunden wird, ist der Rückgabecode 5.

`FINDCHANGE FORWARD/S, BACKWARD/S, CASEON/S, CASEOFF/S, WORDON/S, WORDOFF/S, FROMBLOCK/S, ALL/S, UNTIL/K, PROMPT/S, FIND/K, CHANGE/K`

Sucht nach einem Text und ändert ihn. PROMPT bringt vor dem Austausch einen Frage-Requester hoch. FIND gibt den Suchtext an, CHANGE den Ersetztext. Mit ALL wird nach dem Austausch gleich weitergesucht. Wenn kein Such- oder Ersetzstring angegeben wird, benutzt der Editor die aktuell eingestellten Strings. Nach UNTIL muß ein Markenname folgen. Von der aktuellen Position aus wird gesucht, bis die Marke erreicht ist. Wenn die Marke vor der aktuellen Position liegt und vorwärts gesucht wird, findet der Editor nichts und bricht sofort ab. Nach der Ausführung sind die obigen Einstellungen übernommen. Anmerkung: Das UNTIL-Feature ist vorerst nicht implementiert. Wenn der String nicht gefunden wird, ist der Rückgabecode 5.



`MOVEMATCHBKT`

Findet die Klammer, die zu der unter dem Cursor stehenden paßt, und positioniert den Cursor auf ihr. Ist das Zeichen unter dem Cursor keine Klammer, ist der Rückgabecode 5. Existiert keine passende Klammer, dann gibt es einen Fehler. Als Klammern gelten:

( ) [ ] { } < >

Diese Funktion wird auch durch Doppelklick auf eine der oberen Klammern ausgelöst.

### 2.7.12 ARexx-Befehle zur Statusinformation

`COPYRIGHT`

Öffnet einen Requester mit der Copyright-Meldung und der Angabe des noch freien Speichers. Hier ist auch der ARexx-Hauptportname verzeichnet.

`GETVERSION`

Gibt für ARexx die Versionsnummer des Editors zurück.

`HELP COMMAND/A/F`

Liefert zu einem Befehl die passende Parameter-Schablone zurück. Falls der Befehl nicht existiert, ist der Rückgabecode 5.

`GETCHAR`

Gibt das Zeichen unter dem Cursor zurück.

`GETWORD`

Gibt das Wort unter dem Cursor zurück. Ob Worte komplex sind, ist durch die Preferences bestimmt. Falls der Cursor unmittelbar hinter einem Wort steht, wird dieses trotzdem zurückgegeben.

`GETLINE /N`

Gibt den Inhalt einer Zeile zurück. Wenn die Zeilennummer nicht angegeben ist, wird die aktuelle Zeile zurückgegeben. Falls die Zeile nicht existiert, ist der Rückgabestring leer und der Rückgabecode 5.

`GETCURSORPOS`

Gibt einen String der Form ≫z s≪ zurück, wobei ≫z≪ die Zeilennummer und ≫s≪ die Spaltennummer ist.

`GETDOCUMENTS`

Gibt einen String mit den ARexx-Portnamen der derzeit existierenden Puffer zurück.

**GETPORT MAIN/S FILENAME/F**

Holt den ARexx-Portnamen der angegebenen Datei. Falls FILENAME und MAIN fehlen, wird der Portname des aktuellen Puffers geholt. Der FILENAME muß mit dem im Fenstertitel erscheinenden Namen einschließlich Groß-/Kleinschreibung identisch sein. Ist MAIN angegeben, dann wird der Name des ARexx-Hauptports zurückgegeben und FILENAME ignoriert.

**GETFILEINFO**

Gibt Information für den Puffer zurück. Das Ergebnis hat die Form »zeilen mod Name«, dabei ist »zeilen« die Anzahl der Zeilen, »mod« ist »YES« oder »NO« und zeigt an, ob der Puffer verändert wurde, und »Name« ist der im Fenstertitel sichtbare Name des Puffers.

**GETFILEPATH**

Gibt den kompletten Pfadnamen des Puffers zurück. Falls der Puffer noch nicht als Datei auf Platte existiert, kann dieser Name falsch sein, sofern der Fenstertitel noch Verzeichnisangaben enthält.

**GETSCREENINFO**

Gibt Information über den aktuellen Screen zurück, auf dem der Editor gerade seine Fenster geöffnet hat. Das Ergebnis hat die Form »left top width height depth xclip yclip«. Die Werte sind die Position, die Größe und Tiefe des Screens. Die Clipwerte geben die sichtbare Größe des Screens an und sind bei scrollbaren Screens von den vorderen Größenangaben verschieden.

**GETWINDOWINFO**

Gibt Information über das aktuelle Textpufferfenster zurück. Das Ergebnis hat die Form »hidden x y width height minwidth minheight numviews«. Dabei ist »hidden« gleich »OFF« bzw. »ON«, wenn das Textfenster über den Befehl HIDEFILE unsichtbar gemacht wurde. Die nächsten vier Werte geben die Position und Größe des Fensters an und die nächsten beiden die minimale Größe des Fensters. »numviews« sagt aus, wieviele Editierbereiche derzeit existieren.

**GETVIEWINFO**

Gibt Information über den aktuellen Editierbereich zurück. Das Ergebnis hat die Form »view lines chars topline leftchar«. »view« ist die Nummer des Editierbereiches, »lines« die sichtbare Zeilenanzahl, »chars« die sichtbare Zeichenanzahl, »topline« die Zeilennummer der obersten sichtbaren Zeile des Editierbereiches und »leftchar« die linke, sichtbare Spalte.



### 2.7.13 ARexx-Befehle zur Textänderung

**TEXT TEXT/F**

Fügt den angegebenen Text an der aktuellen Position ein. Dabei werden Overstrike und Wordwrap berücksichtigt.

**INSERT TEXT/F**

Synonym zu TEXT.

**FORMATPARAGRAPH**

Formatiert einen Absatz wie im Wordwrap-Modus, ohne Zeichen einzufügen oder zu entfernen.

**TABINDENT BACK/S**

Rückt den Text der aktuellen Zeile nach rechts, und zwar auf die nächste Position, an der in der darüberstehenden Zeile das nächste Wort beginnt, wobei hier Worte nur durch Leerzeichen und Tabulatorzeichen getrennt werden, d.h., die Preference-Einstellung »Complex Words« gilt hier nicht. Das Einrücken geschieht durch Einfügen von Leerzeichen (der Overstrike-Modus hat hier keine Auswirkung). Gibt es den Wortanfang oder die vorige Zeile nicht, dann wird zur nächsten Tabulatorposition eingerückt. Mit BACK wird rückwärts gearbeitet, was allerdings beim Löschen von Tabulatorzeichen zu falschen Ergebnissen führen kann (dies ist von Hand zu korrigieren). Es werden nur soviel Leer- und Tabulatorzeichen gelöscht wie möglich, übrige Zeichen bleiben erhalten.

**NEWLINE**

Spaltet die Zeile an der aktuellen Cursorposition und bewegt den Zeilenrest in eine neu erzeugte Zeile. Der Zeilenrest beginnt auf der neuen Zeile erst am linken Rand.

**NEWLINEINDENT**

Wie NEWLINE, aber statt am linken Rand wird in der neuen Zeile auf derselben Position begonnen, an der sich in der alten das erste Zeichen befindet, das weder ein Leer- noch ein Tabulatorzeichen ist.

**APPENDLINE**

Bewegt den Cursor ans Zeilenende und hängt die nächste Zeile an die aktuelle an. Gibt es keine Nachfolgerzeile, ist der Rückgabecode 5.

**DELETELINE**

Löscht die aktuelle Zeile und geht auf die Folgezeile. Gibt es diese nicht, wird die aktuelle nur geleert, nicht entfernt, und der Rückgabecode ist 5. Der alte Zeileninhalt kann mit UNDELETELINE zurückgeholt werden.

**UNDELETELINE**

Holt den Inhalt des letzten DELETELINE zurück. Dabei wird unabhängig von der Cursorposition immer die komplette Zeile vor der aktuellen eingefügt.

**DELETE CHARS/N**

Löscht soviele Zeichen wie angegeben unter dem Cursor, ohne ihn zu bewegen. Keine Angabe löscht ein Zeichen. Wenn WrapAround durch SETPREFS eingeschaltet ist, führt DELETE am Zeilenende zu APPENDLINE. CHARS kann maximal 254 sein.

**BACKSPACE**

Löscht das Zeichen vor (links von) dem Cursor und bewegt ihn nach links. Wenn der Cursor auf Spalte 0 stand, wird die aktuelle Zeile an die vorige per APPENDLINE angehängt.

**UNDOLINE**

Stellt den Zeileninhalt wieder so her, wie er beim Anwählen der Zeile war. Im Wordwrap-Modus kann dies wegen der ständig stattfindenden Formatierung zu unerwarteten Ergebnissen führen.

**SETCHAR CHAR/A**

Setze das Zeichen unter dem Cursor auf CHAR. Hat CHAR mehrere Zeichen, dann wird nur das erste genommen.

**EXCHANGECHARS**

Vertauscht das Zeichen unter dem Cursor mit dem hinter (rechts von) ihm.

**SWAPCHARS**

Vertauscht das Zeichen unter dem Cursor mit dem vor (links von) ihm.

**TOGGLECHARCASE**

Wandelt das Zeichen unter dem Cursor von Groß- nach Kleinschreibung bzw. umgekehrt und geht einen Schritt nach rechts.

**INSERTASCII STRING/F**

Fügt ein ASCII-Zeichen per Code-Angabe in den Text ein. STRING muß eine Dezimalzahl oder Hexadezimalzahl im Bereich 0 bis 255 sein. Falls STRING nicht angegeben ist, wird auf den interaktiven Modus geschaltet, in dem so lange per Requester Zeichen eingefügt werden, bis ein leerer String als Eingabe kommt oder Cancel angeklickt wird.

**INSERTSPACE**

Fügt ein Leerzeichen an der aktuellen Cursorposition ein, auch wenn der Overstrike-Modus eingeschaltet ist.

**DELETESOL**

Löscht alle Zeichen vom Zeilenbeginn bis zum Cursor. Das Zeichen unter dem Cursor bleibt unberührt. SOL heißt Zeilenanfang (Start-of-line). Der Cursor steht nach diesem Befehl auf dem Zeilenanfang.

**DELETEEOL**



Löscht alle Zeichen unter bzw. hinter dem Cursor bis zum Zeilenende. EOL heißt Zeilenende (End-of-line).

**CLEARSOL**

Setzt alle Zeichen vom Zeilenbeginn bis zum Cursor auf Leerzeichen. Wenn in den Preferences »AutoSOLTab« eingeschaltet ist, dann werden entsprechend auch Tabulatorzeichen gesetzt. Das Zeichen unter dem Cursor bleibt unberührt.

**DELETESOW**

Wenn Cursor auf einem Wort steht, werden alle Zeichen vor dem Cursor bis zum Wortanfang gelöscht, sonst alle Zeichen bis zum Ende des vorigen Wortes.

**DELETEEOW**

Wenn Cursor auf einem Wort steht, werden alle Zeichen vom Cursor an bis zum Wortende gelöscht, sonst alle Zeichen bis zum Anfang des nächsten Wortes.

**REPLACEWORD WORD/F/A**

Ersetzt das Wort unter dem Cursor durch das angegebene Wort. Falls der Cursor auf keinem Wort steht, wird das Wort eingefügt.

**UPPERCASE CHAR/S, WORD/S, LINE/S, BLOCK/S**

Ändert alle Zeichen des angegebenen Bereiches auf Großbuchstaben. Der Editor wandelt auch deutsche bzw. anderssprachige Sonderzeichen um. So wird z.B. aus »ö« ein »Ö« oder aus »æ« ein »Æ« etc.

**LOWERCASE CHAR/S, WORD/S, LINE/S, BLOCK/S**

Ändert alle Zeichen des angegebenen Bereiches auf Kleinbuchstaben. Der Editor wandelt auch deutsche bzw. anderssprachige Sonderzeichen um. So wird z.B. aus »Ö« ein »ö« oder aus »Æ« ein »æ« etc.

**SWAPCASE CHAR/S, WORD/S, LINE/S, BLOCK/S**

Ändert alle Zeichen des angegebenen Bereiches von Groß- auf Kleinbuchstaben bzw. umgekehrt. Der Editor wandelt auch deutsche bzw. anderssprachige Sonderzeichen um. So wird z.B. aus »ö« ein »Ö« oder aus »Æ« ein »æ« etc.

**CONV2UPPER**

Ändert den markierten Block (bzw. das aktuelle Wort, wenn kein Block markiert ist) auf Großschreibung.

**CONV2LOWER**

Ändert den markierten Block (bzw. das aktuelle Wort, wenn kein Block markiert ist) auf Kleinschreibung.

### 2.7.14 ARexx-Befehle für Assembler und Fehlerliste

**OPENASSEMBLE**

Öffnet den Assemble-Requester. Der aktuelle Puffer kann nachher assembliert werden. Im Object-Gadget kann der Name der zu erzeugenden Objektdatei angegeben werden, und im Options-Gadget können die Assembler-Optionen angegeben werden, die nicht durch die Checkboxen erreichbar sind. Die Einstellungen des Assemble-Requesters werden in der Preferencedatei abgespeichert.

**ASSEMBLE**

Assembliert einen Puffer mit den aktuellen Einstellungen des Assemble-Requesters. Wenn der Befehl über den Hauptport kam, wird der aktuelle Puffer assembliert, sonst der angesprochene. Die Assemblierung startet sofort, ohne den Assemble-Requester aufzurufen. Ich empfehle, statt OPENASSEMBLE und ASSEMBLE das Make-Utility zu benutzen. Dann wird der Assembler nur durch einen Tastendruck gestartet, und es stehen all die mächtigen Möglichkeiten von MAKE zur Verfügung.

**RUNPROGRAM**

Läßt das vom Assembler in den Speicher assemblierte Programm ablaufen. Falls Ihr Programm abstürzt, hängt sich aber auch der Editor auf!

**LOADASSEMBLER FILENAME/F**

Lädt den Optimierenden Makro-Assembler resident. Falls der Assembler bereits resident ist oder die zu ladende Datei nicht der OMA ist, gibt es eine Fehlermeldung. Falls FILENAME nicht angegeben ist, erscheint der Filerequester zur Auswahl. Falls Cancel gewählt wird oder der Name leer ist, ist der Rückgabecode 5. Der angegebene Name wird als Assembler-Defaultname in die Preferences übernommen.

**UNLOADASSEMBLER**

Entfernt den residenten Assembler, die Fehlerliste und das vom Assembler in den Speicher assemblierte Programm.

**FIRSTERROR**

Wählt den ersten Fehler der Fehlerliste an. Wenn kein Eintrag in der Fehlerliste existiert, dann gibt es einen Fehler mit Rückgabecode 20.

**NEXTERROR**

Wählt den nächsten Fehler der Fehlerliste an. Wenn kein nächster Eintrag existiert, gibt es einen Fehler mit Rückgabecode 10. Wenn keine Fehlerliste existiert, gibt es einen Fehler mit Rückgabecode 20.

**PREVERROR**

Wählt den vorigen Fehler der Fehlerliste an. Wenn kein nächster Eintrag existiert, gibt es einen Fehler mit Rückgabecode 10. Wenn keine Fehlerliste existiert, gibt es einen Fehler mit Rückgabecode 20.



**CURRENTERROR**

Wählt den aktuellen Fehler der Fehlerliste an. Wenn kein nächster Eintrag existiert, gibt es einen Fehler mit Rückgabecode 10. Wenn keine Fehlerliste existiert, gibt es einen Fehler mit Rückgabecode 20.

**READERRORLIST FILENAME/F**

Lädt eine Datei mit Fehlerlisten-Inhalt. Damit können Fehler eines Compilers bzgl. eines Puffers bequem über NEXTERROR angesprungen werden. Vorteil: Beim Einfügen von Zeilen u.ä. wird die Fehlerstelle immer korrekt mitgeführt. Falls FILENAME nicht angegeben ist, erscheint der Filerequester zur Auswahl. Falls Cancel gewählt wird oder der Name leer ist, ist der Rückgabecode 5. Das Format der Fehlerlistendatei ist:

```
; Dies ist ein Kommentar.
zeile spalte Fehlertext
```

»Zeile« ist die Zeilennummer, »Spalte« die Zeichenposition innerhalb der Zeile, wobei das erste Zeichen Position 0 hat und Tabulatoren als ein Zeichen zählen. Wenn »Fehlertext« Leerzeichen u.ä. enthalten soll, muß er in Hochkommata eingefaßt werden. Der Backslash ist das Fluchtsymbol. Es sind zwar beliebig viele Fehler möglich, aus Performancegründen sollte deren Anzahl aber nicht zu groß werden. Die Fehlerliste gehört zu dem Puffer, an dessen Port der Befehl geschickt wurde (Hauptport bedeutet aktueller Puffer). Siehe auch Kapitel 2.8 über das Make-Utility.

**CLEARERRORLIST**

Löscht die Fehlerliste und das vom Assembler in den Speicher assemblierte Programm. Dies ist nützlich, wenn sie sehr lang ist und bei Pufferänderungen Performance-Einbußen verursacht.

## 2.8 Das Make-Utility

### 2.8.1 Beschreibung

Der Befehl MAKE startet das integrierte Make-Utility. MAKE ermöglicht die einfache Behandlung von großen Softwarepaketen, indem nach einmaliger Erzeugung eines Makefiles und damit der Angabe der Abhängigkeiten der Module nur MAKE aufgerufen werden muß. Dann werden alle nötigen Aktionen ausgeführt, um zu der neuesten, gültigen Programmversion zu gelangen. Im Makefile sind die Abhängigkeiten zeilenweise angegeben. Es hat die folgende Syntax:

- Eine Leerzeile wird ignoriert.
- Kommentarzeilen beginnen mit einem Hash-Zeichen »#« oder Semikolon »;« als erstem Zeichen und enden mit dem Zeilenende. Ein Kommentar darf ferner in jeder Zeile enthalten sein, wenn er wie bei einer Shell mit einem Semikolon beginnt; davor muß jedoch mindestens ein Leer- oder Tabulatorzeichen stehen. Der Kommentar ist bis zum Zeilenende gültig.
- Eine oder mehrere Abhängigkeitszeilen gehen einer Reihe von Kommandozeilen voraus. Die Kommandozeilen beginnen mit einem Leer- oder Tabulatorzeichen und werden wie von einer Shell ausgeführt, wenn mindestens eine der Abhängigkeitszeilen das Resultat TRUE liefert. Vor der Ausführung einer Kommandozeile gibt MAKE die Kommandozeile mit dem Vorsatz »Make: « aus, so daß die Aktionsfolge dokumentiert wird.
- Abhängigkeitszeilen haben die folgende Syntax:

  ```
  HAUPTDATEI: DATEI1 DATEI2 DATEI3 etc.
  ```

  Die Dateinamen werden durch Leer- oder Tabulatorzeichen getrennt; der erste Dateiname muß mit einem Doppelpunkt enden und auf dem ersten Zeichen der Zeile beginnen. Das Resultat der Zeile ist TRUE, wenn die Hauptdatei nicht existiert oder eine der darauf folgenden Dateien (diese heißen Abhängigkeitsdateien) ein jüngeres Datum als die Hauptdatei hat. Wichtige Voraussetzung dazu ist, daß die Systemuhr richtig gesetzt ist. Nehmen Sie einen Hauptdateinamen, der nicht existiert (evtl. sinnvollerweise true:), so ist das Ergebnis der Abhängigkeitszeile immer TRUE, und die folgenden Kommandozeilen werden immer ausgeführt. Steht der Hauptdateiname alleine, ist das Ergebnis ebenfalls TRUE.

  Es muß erwähnt werden, daß der Editor für eine Abhängigkeitszeile auch dann TRUE setzt, wenn eine Abhängigkeitsdatei in einem Textpuffer steht und dieser verändert wurde. Es wird aber nur geprüft, ob die Abhängigkeitsdatei und der Puffername dasselbe Namensende haben. Falls das Verzeichnis unterschiedlich ist, wird dies eventuell nicht erkannt (siehe auch folgenden Abschnitt). Diese Einschränkung ist für den residenten Assembler wichtig.

Beispiel: Das Makefile lautet

```
# Dies ist ein Kommentar
make.obj: make.asm oma make.asm
```



D.h.: make.obj hängt von make.asm ab. Wenn make.obj nicht existiert oder make.asm ein jüngeres Datum als make.obj hat, dann assembliere make.asm.

Beispiel:

```
make.o:   make.c
make.o:   make.h
   cc make.c
```

und make.c inkludiert make.h. In dem Fall wird make.o produziert, falls make.c oder make.h jünger als make.o ist oder make.o nicht existiert. Da mehrere Abhängigkeitsdateien in einer Zeile angegeben werden können, lautet das Beispiel verkürzt

```
make.o:   make.c   make.h
   cc make.c
```

Statt cc make.c können mehrere Kommandozeilen stehen, z.B.

```
   cc make.c
   ln make.o -lc
```

Die Kommandozeilen sind aber voneinander unabhängig, ein Verzeichniswechsel mit dem Shell-Befehl CD ist also in der folgenden Kommandozeile nicht mehr gültig. Für diese Fälle kann die Kommandozeile Execute Batchfile mit dem entsprechenden Inhalt von Batchfile als Ersatz benutzt werden; in Batchfile hat der CD-Befehl dann Shell-gemäß seine Auswirkung auf den DOS-Prozeß.

Beim Aufruf des Befehls MAKE kann der Name des zu verwendenden Makefiles angegeben werden; falls er fehlt, lautet er makefile. Der Editor liest das Makefile aus einem Puffer, wenn der Name des Puffers mit dem Makefilenamen endet, z.B. wie bei mein.makefile. Ansonsten wird nach einer Datei im Verzeichnis des aktuellen Puffers (bzw. im Heimatverzeichnis des Editors, falls gerade kein Puffer existiert) gesucht, die den Namen des Makefiles hat, und diese als Makefile gelesen. Da beim Lesen aus dem Puffer nur die Endung des Puffernamens mit dem Makefilenamen gleich sein muß, können Sie in einem Verzeichnis mehrere Programme mit ihrem Makefiles unterbringen und die Makefiles entsprechend benennen, z.B. die Programme hole.asm und cls.asm und ihre Makefiles hole.makefile und cls.makefile. Laden Sie das Makefile dann in einen Puffer, so wird es von MAKE bearbeitet werden. Beachten Sie, daß Sie dann notwendigerweise das Makefile in einen Puffer laden müssen, da MAKE ohne Argument den Namen makefile im aktuellen Verzeichnis sucht. Beachten Sie ferner, daß der Editor den ersten Puffer in der Pufferliste nimmt, der das Endungskriterium erfüllt. Ähnliches gilt für Textpuffer, die als Abhängigkeitsdateien direkt erkannt werden (siehe oben).

MAKE schaltet vom Editor-Screen auf die Workbench um, öffnet sein Ausgabefenster und arbeitet das Makefile ab, was durch <Ctrl>+<D> unterbrochen werden kann. Danach wartet MAKE auf das Drücken der Return-Taste und schaltet zurück auf den Editor. Die von MAKE ausgeführten Aktionen lassen sich selber nicht z.B. durch

<Ctrl>+<C> abbrechen (Ausnahme: der residente Assembler), der Abbruch erfolgt nur beim Wechsel von Makefile-Zeilen.

Sie können natürlich gerade assemblierte bzw. gelinkte Programme von MAKE ausführen lassen, um sie zu testen. Im folgenden Beispiel wird das Programm nur bei Bedarf neu assembliert, aber bei jedem Aufruf von MAKE gestartet:

```
ram:prg: prg.asm
   oma prg.asm -o ram:prg
true:                ; immer TRUE, wenn Datei "true" nicht existiert (s.o.)
   run ram:prg
```

Falls Sie wenig Speicher haben, sollten Sie das Hilfsprogramm Make benutzen, das bis auf die Unterstützung des residenten Assemblers die gleichen Möglichkeiten wie das im Editor eingebaute MAKE bietet.

### 2.8.2 Der residente Assembler

Eine Besonderheit existiert beim Make-Utility im Zusammenhang mit dem Optimierenden Makro-Assembler OMA. Wenn OMA mittels LOADASSEMBLER resident ist und eine Kommandozeile den Befehl OMA (unabhängig von Groß-/Kleinschreibung) enthält oder aber über die Befehle OPENASSEMBLE oder ASSEMBLE assembliert werden soll, dann wird der residente Assembler gestartet. Er ist dann in der Lage, seine Quelldateien aus den Editor-Puffern zu holen, was sehr schnell ist, eine Fehlerliste zu erzeugen sowie Programme in den Speicher zu assemblieren, um sie unter dem Editor ausführen zu lassen.

Will der Assembler eine Quelltextdatei öffnen, prüft er nur die Endungen der Puffernamen und beachtet nicht, ob der betreffende Text aus dem richtigen Verzeichnis stammt. So wird z.B. die Datei make.asm aus dem Directory df0:examples auch aus dem Puffer namens text/make.asm gelesen, dessen Directory z.B. df1: lautet. Falls mehrere Puffer in Frage kommen, wird der in der Pufferliste zuerst vorkommende Puffer genommen.

Der MAKE-Lauf wird abgebrochen, wenn der Assembler durch <Ctrl>+<C> unterbrochen (dies funktioniert nur beim residenten Assembler) oder ein Fehler im Quelltext gefunden wurde. Dann sind die Fehlerlisten-Befehle FIRSTERROR etc. im eingeschaltet. Sobald Sie vom Make-Utility zurückkehren, wird die erste fehlerhafte Zeile über den Befehl NEXTERROR angesteuert.

Bei der Fehleranwahl schaltet der Editor, falls die Datei in einem Puffer steht, auf den Puffer, springt die fehlerhafte Zeile an und meldet den Fehler in der Statuszeile. Danach kann normal weitergearbeitet und der Fehler entfernt werden. Falls die Datei nicht in einem Puffer steht, führt der Editor OPENNEWFILE aus und lädt dadurch die Datei ein. Danach wird die Fehlerstelle angesprungen.

Diese Behandlung von Fehlern ermöglicht das schnelle Erstellen von assemblierbaren Programmen, da der Quelltext so lange im Editor gehalten werden kann, bis der Assemblerlauf erfolgreich ist. Die Fehlerliste selber wird bei jedem MAKE-Start und bei den



Befehlen CLEARERRORLIST und UNLOADASSEMBLER gelöscht. Über ARexx-Programme und den Befehl READERRORLIST ist es möglich, die Fehlerlisten des Editors für jeden beliebigen Compiler oder Assembler zu benutzen.

### 2.8.3 Assemblieren in den Speicher

Der Optimierende Makro-Assembler unterstützt das Assemblieren in den Speicher. Damit können Sie bei Quelltexten, die gleich ausführbare Programme ergeben, das Programm gleich unter dem Editor starten und kontrollieren. Schalten Sie dazu das »Into memory«-Gadget des Assemble-Requester ein und assemblieren Sie normal. Neben der normalen Objektdatei-Produktion schreibt der Assembler das Programm dabei auch in den Speicher. Nachdem Sie in den Editor zurückgekehrt sind, können Sie das Programm gleich über den Befehl RUNPROGRAM starten. Dabei können Sie dem Programm allerdings keine Shell-Parameter übergeben, und bei Programmabsturz bleibt auch der Editor hängen, und die Pufferinhalte sind verloren! Zudem kann der Debugger DBug dabei nicht benutzt werden.

Das residente Programm wird vor jedem Assemblerstart und beim Befehl UNLOADASSEMBLER gelöscht.

Das Assemblieren in den Speicher steht auch beim Make-Utility für das jeweils zuletzt assemblierte Programm zur Verfügung, wenn Sie bei dem Assembleraufruf die Startoption »-c z« angeben. Beachten Sie aber, daß für das erfolgreiche Assemblieren in den Speicher und Starten des residenten Programmes wie oben vier Bedingungen zutreffen müssen:

1. Der Assembler muß resident geladen sein.
2. Die »-c z«-Startoption muß benutzt werden.
3. Der Quelltext muß ein ausführbares Programm ergeben, kein linkbares.
4. Der Assembler darf keine Fehler finden.

Sollte für den Befehl RUNPROGRAM kein residentes Programm vorliegen, meldet der Editor entsprechend einen Fehler.

## 2.9 Die Preferences

Die Preferences des Editors sind Einstellungen bestimmter Parameter, die die Arbeitsweise des Editors beeinflußen und während der Laufzeit verändert werden können. Dazu gibt es zwei Requester, mit denen die Werte bequem über Gadgets manipulierbar sind. Die Preferences lassen sich mit SAVEPREFS abspeichern und mit OPENPREFS oder beim Start einladen (siehe 2.9.3 folgend). Wird keine Preferencedatei gelesen, wird eine Standardeinstellung verwandt. Die meisten der einzelnen Werte können auch über die Befehle SETPREFS geschrieben und über GETPREFS gelesen werden.

Bitte beachten Sie bei den Requestern, daß bei Stringeingaben der neue Wert nur akzeptiert wird, wenn Sie nach der Eingabe die Return-Taste drücken. Der Wechsel auf ein anderes Gadget durch Anklicken genügt nicht.

### 2.9.1 Der Standard-Preference-Requester

Im Standard-Preference-Requester werden normale Editiereinstellungen vorgenommen. Der Requester erscheint auf den Befehl OPENPREFWINDOW hin und besitzt die folgenden Gadgets:

**Wrap Around:** Soll der Cursor am Zeilenanfang/-ende auf die vorige/nächste Zeile springen, wenn z.B. "CURSOR right" am Zeilenende befohlen wird? Dies gilt auch für "NEXT/PREVIOUS word" und andere Befehle. Die Standardeinstellung ist OFF.

**Save Icons:** Sollen Icons für zu speichernde Texte geschrieben werden? Für Texte, die schon beim Einladen ein Icon besaßen, wird automatisch wieder dieses Icon geschrieben, auch wenn diese Option abgeschaltet ist. Unbekannte Tooltypes läßt der Editor im Icon unverändert. Siehe auch dazu Kapitel 2.3.2. Die Standardeinstellung ist OFF.

**Auto SOL Tabs:** Wenn diese Checkbox markiert und TabExpand eingeschaltet ist, dann werden vom Zeilenanfang an alle Leer- und Tabulatorzeichen bis zum ersten Zeichen, das weder ein Leer- noch Tabulatorzeichen ist, automatisch durch die minimale Anzahl von Tabulator- und Leerzeichen aufgefüllt. Dies funktioniert dann bei allen Funktionen. Die Standardeinstellung ist ON.

**Complex Words:** Normalerweise gehört alles zu einem Wort, was weder Leer- noch Tabulatorzeichen ist. Ist diese Checkbox jedoch markiert, dann werden Wörter auch durch Zeichen wie Komma, Punkt, Klammern etc. getrennt. Nur Buchstaben, Ziffern und der Unterstrich zählen dann zu einem Wort. Diese Einstellung wird bei allen wortorientierten Befehlen berücksichtigt. Die Standardeinstellung ist OFF.

**Tab Width, Left Margin, Right Margin** sind der Tabulatorabstand, der linke Rand und der rechte Rand des aktuellen Textpuffers. Die beiden Ränder werden vertauscht, falls durch die Wahl der rechte Rand links vom linken liegt. Die Standardeinstellungen sind 8, 0 und 79. Die Ränder legen z.B. fest, wo für den Befehl NEWLINE die Zeile beginnt und in welchen Grenzen der Wordwrap-Modus arbeitet.



**Global Tab, Global Left, Global Right** sind entsprechend die Werte, mit denen ein mit dem Befehl NEWFILE oder OPENNEWFILE neu erzeugter Puffer anfängt. Die Standardeinstellungen sind 8, 0 und 79.

**Backup Mode:** Der Editor kennt verschiedene Methoden, eine Sicherungskopie (Backup) einer vorher schon existierenden Datei beim Schreiben eines Puffers per SAVEFILE zu erzeugen. Dies ist nur möglich, falls eine alte Datei gleichen Namens existiert. Falls kein Backup erstellt werden konnte, meldet der Editor dies. Die Standardeinstellung ist OFF. Es gibt drei verschiedene Backup-Modi:

1. Es wird kein Backup erzeugt (≫OFF≪).
2. Falls im Wurzelverzeichnis des Volumes, von der der bearbeitete Text stammt, ein T-Verzeichnis existiert, wird der Editor die alte Datei in HiTex-Backup umbenennen und sie in dieses T-Verzeichnis verlagern. Ein dort vor dem Schreiben vorhandenes Backup geht verloren. Mit diesem Modus haben Sie nur vom zuletzt geschriebenen Text ein Backup (≫T≪).
3. Der Editor benennt die Datei in einen Namen um, der sich aus dem alten Namen und dem Suffix ≫.bak≪ zusammensetzt, und läßt es im gleichen Verzeichnis. Falls der Name mit ≫.asm≪ endet, wird dieses Suffix durch ≫.bak≪ ersetzt. Ein schon vorhandenes Backup mit dem Namen geht verloren. Mit diesem Modus haben Sie von jeder geschriebenen Datei ein eigenes Backup (≫BAK≪).

**Default Name:** Dies bestimmt einen Dateinamen, unter dem der Optimierende Makro-Assembler gleich beim Editorstart eingeladen werden kann. Beachten Sie, daß dieser Name abgeändert wird, wenn Sie über den Befehl LOADASSEMBLER einen anderen Namen für den Assembler wählen. Falls beim Editorstart der Assembler-Defaultname angegeben und nicht leer ist, versucht der Editor, diese Datei als residenten Assembler zu laden. Dadurch haben Sie Ihre Assembler-Programmierumgebung sofort parat. Die Standardeinstellung ist ein leerer Name.

**Console:** Dies ist der Dateiname für ein Consolefenster für das Make-Utility und die Befehle RUNPROGRAM, OPENASSEMBLE, ASSEMBLE, EXECAREXXMACRO und EXECAREXXSTRING. Die Standardeinstellung ist CON:0/0/640/200/Make.

**File Pattern:** Dies ist das Dateimuster, das der Filerequester benutzt. Bitte beachten Sie, daß das Ändern des Patterns im Filerequester diesen Wert hier auch verändert und umgekehrt. Die Standardeinstellung ist `(~(#?.info))`, d.h. alle Icon-Dateien ausblenden.

Das **Ok-Gadget** schließlich beendet den Requester.

### 2.9.2 Der Screen-Preference-Requester

Dieser Requester enthält Einstellungen, die den Screen wählen, auf dem der Editor arbeitet und seine Parameter einstellt. Bei den meisten Einstellungen wird es nötig, alle

Fenster zu schließen und erneut zu öffnen und u.U. auch den Screen erneut einzuschalten. Der Requester erscheint auf den Befehl OPENSCREENPREFWINDOW hin und besitzt die folgenden Gadgets:

**Smart Refresh:** Wenn eingeschaltet, werden neu geöffnete Fenster mit dem Smart-Refresh von Intuition ausgestattet. Der Refresh ist dann sehr schnell, benötigt aber eine relativ hohe Speichermenge. Ausgeschaltet bekommen die Fenster Intuitions Simple-Refresh. Dieser braucht sehr wenig Speicher, dauert aber länger und bedingt bei einigen Editorfunktionen eine höhere Laufzeit (z.B. beim Ziehen eines Scrollbars). Diese Einstellung gilt für Textfenster, Hilfsfenster und Requester. Die Standardeinstellung ist »REFRESH SIMPLE«

**Depth:** bestimmt die Tiefe des eigenen, vom Editor geöffneten Screens. Siehe auch den folgenden Punkt.

**Screen Name:** Hier kann ein Screenname eingegeben werden. Wenn bereits ein Public-Screen mit gleichem Namen existiert, öffnet der Editor seine Fenster auf diesem Screen (kein Name heißt Workbench). Sonst öffnet er einen eigenen, privaten Screen und öffnet seine Fenster dort. Der zu besuchende Screen sollte nach Möglichkeit mindestens vier Farben haben. Die Standardeinstellung ist ein leerer Name, d.h., der Editor arbeitet auf der Workbench.

**Screen Width, Screen Height** sind die Breite und Höhe des eigenen Screens in Pixeln. Diese Werte werden automatisch bei der Videomodus-Wahl gesetzt.

**Video Mode** wählt einen Videomodus, wie er auch im System-Preference-Programm Screenmode ausgewählt werden kann. Der Videomodus gilt für den eigenen Screen. Der in einer Preferencedatei gewünschte Videomodus sollte auf dem Rechner vorhanden sein, sonst startet der Editor nicht richtig, sondern mit dem Standardmodus.

**Die Gadgets zur Farbauswahl** gelten ebenfalls nur für den eigenen Screen. Der eigene Screen bekommt den angewählten Videomodus und die angegebene Farbeinstellung. »WB Colors« setzt die Farben auf die aktuelle Einstellung der Workbench, und »Reset Colors« setzt die Farben auf die alten Werte zu dem Zeitpunkt, zu dem der Screen-Preference-Requester geöffnet wurde. Es können maximal vier Farben eingestellt werden.

**Select Font...:** Hier erscheint ein System-Fontrequester, mit dem der Font bestimmt werden kann, der zur Textdarstellung in den Editierbereichen der Textfenster benutzt werden soll. Die Standardeinstellung ist der aktuelle System-Default-Font, der durch das System-Preference-Programm Font gesetzt wurde.

### 2.9.3 Die Preferencedatei

Wie im Kapitel 2.2 beschrieben wurde, werden die Einstellungen beim Start aus einer Datei gelesen. Der Editor versucht zuerst, eine Datei namens HiTex.prefs zu holen. Gelingt dies nicht, probiert er es noch einmal mit der Datei ACOM:HiTex.prefs. Falls diese auch nicht existiert, dann werden die internen Voreinstellungen benutzt. Dieser Leseprozeß kann mit der PREFS-Startoption auf eine feste Datei gelenkt werden. Ist diese



Startoption angegeben, dann werden die Einstellungen nur aus dieser Datei gelesen. Das Format der Preferencedatei ist wie folgt:

```
; Dies ist ein Kommentar.
REFRESH [ SIMPLE | SMART ]
SCREEN NAME WIDTH HEIGHT VIDEOMODE
DEPTH N
COLOR0 R G B
COLOR1 R G B
COLOR2 R G B
COLOR3 R G B
FONT NAME SIZE
TABWIDTH WIDTH                          ; der globale Wert
LEFTMARGIN N                            ; der globale Wert
RIGHTMARGIN N                           ; der globale Wert
OPENSIZE WIDTH HEIGHT                   ; Öffnungsgröße für Textfenster
SAVEICONS [ ON | OFF ]
BACKUPMODE [ OFF | T | BAK ]
DEFAULTNAME FILENAME                    ; Dateiname des Assemblers
CONSOLENAME CONSOLENAME                 ; Dateiname der Console für Make u.ä.
AUTOSOLTABS [ ON | OFF ]
COMPLEXWORDS [ ON | OFF ]
HIDEMOUSE [ ON | OFF ]                  ; Mauszeiger bei Taste abschalten?
ASLREQUEST [ ON | OFF ]                 ; Standard-Filerequester benutzen?
WRAPAROUND [ ON | OFF ]
FILEPATTERN PATTERN
SEPARATOR WIDTH                         ; Höhe der Editierbereich-Trennzeile
ASSEMBLE OBJECTSTRING OPTIONSTRING FLAGS
```

Hat die Datei beim Lesen nur einige Einstellungen statt allen Angaben, werden auch nur diese Einstellungen abweichend von der alten Einstellung gesetzt. Die Datei kann manuell als Textdatei oder über den Befehl SAVEPREFS erzeugt werden. Die Einstellungen sind dieselben wie oben beschrieben, dazu gibt es noch einige zusätzliche:

**OPENSIZE:** Bestimmt, mit welcher Anfangsgröße sich Textfenster öffnen sollen. Ist die Breite WIDTH negativ, dann werden die Textfenster so groß geöffnet, wie es der sichtbare Screenbereich zuläßt. D.h. bei scrollbaren Screens, daß das Fenster niemals größer als der sichtbare Bereich wird. Die Größe kann auch durch den Befehl FETCHOPENSIZE gesetzt werden. Die Standardeinstellung ist `WIDTH = -1`.

**HIDEMOUSE:** Bestimmt, ob die Maus abgeschaltet werden soll, wenn eine Taste gedrückt wird. Dies ist auf normalen Amigas zu bevorzugen, da dann der Mauszeiger eingegebenen Text nicht verdecken kann. Die Standardeinstellung ist ON. Wenn eine Grafikkarte benutzt wird, mit der sich Probleme im Zusammenhang mit dem Mauswegschalten ergeben, dann sollte hier OFF eingestellt werden.

**ASLREQUEST:** Wie im Kapitel über den Filerequester beschrieben, wählt diese Einstellung aus, ob der System-Standard-Filerequester verwendet wird (ON) oder der dem Editor eigene Filerequester. Die Standardeinstellung ist OFF.

**SEPARATOR:** Die Pixelhöhe der Trennzeilen zwischen Editierbereichen in Textfenstern. Die Standardeinstellung ist vier.

**ASSEMBLE:** Hier ist die Einstellung des Assemble-Requesters abgespeichert. Die einzelnen Flagstellen sind Nullen oder Einsen in der Reihenfolge der Flags im Assemble-Requester.



## 2.10 Konfiguration des Editors

Eine wesentliche Eigenschaft des Editors ist die freie Konfigurierbarkeit von Menüs und Tastaturkommandos. Beim Start des Editors versucht er, eine Textdatei namens HiTex.setup aus dem aktuellen Verzeichnis zu lesen (bzw. beim Workbenchstart aus dem Verzeichnis, aus dem er geladen wurde). Gelingt dies nicht, probiert er es noch einmal im ACOM:-Verzeichnis mit dem gleichen Namen. Schlägt auch das fehl, wird die interne Standardkonfiguration benutzt. Sollte eine Setup-Datei über die SETUP-Startoption gewählt worden sein, so hat sie Vorrang vor den oben genannten Versuchen.

Die Setup-Datei besteht aus zwei Teilen. Der erste Teil konfiguriert die Menüeinstellung, der zweite die Tastaturkommandos.

### 2.10.1 Menükonfiguration

Um die Syntax der Menükonfiguration zu erläutern, ist hier ein Ausschnitt aus der Standardkonfiguration abgedruckt:

```
TITLE Project
 ITEM New NewFile N
 ITEM Open... OpenFile O
 ITEM "Open New..." OpenNewFile
  BAR
 ITEM Save SaveFile S
 ITEM "Save as..." SaveFileAs A
 ITEM "Save modified files" SaveModifiedFiles
 ITEM "Save & Quit" SaveFileQuit
  BAR
 ITEM Rename... SetFilePath
 ITEM "Set home dir..." SetHomeDir
 ITEM Print PrintFile
  BAR
 ITEM Hide HideFile
 ITEM Clear ClearFile
 ITEM Close CloseFile
  BAR
 ITEM Make... Make 5
  BAR
 ITEM Quit Quit Q

TITLE Block
 ITEM Cut CutBlk X
 ITEM Copy CopyBlk C
 ITEM Paste Paste V
 ITEM "Unmark block" UnmarkBlk K
  BAR
```

```
ITEM "Delete block" DeleteBlk
ITEM "Save block" SaveBlk
 BAR
ITEM "Shift left" ShiftLeft
ITEM "Shift right" ShiftRight
 BAR
ITEM Case
 SUB Upper "UpperCase Block"
 SUB Lower "LowerCase Block"
 SUB Swap "SwapCase Block"
 BAR
ITEM Expand Conv2Spaces
; und so weiter. Dies hier ist übrigens ein Kommentar.

; Der Menüteil muß mit folgendem Schlüsselwort beendet werden:
MENUEND
```

Es sind Leerzeilen sowie Kommentarzeilen erlaubt. Ein Kommentar wird an dem Semikolon als erstem Zeichen erkannt. Wie leicht ersichtlich ist, müssen alle Worte, die Leer- oder Tabulatorzeichen enthalten sollen, in Hochkommata eingefaßt werden, die anderen dürfen optional Hochkommata haben. Dabei ist sowohl ein einfaches ' als auch ein doppeltes Hochkomma " möglich. In den Texten ist der Backslash das Fluchtsymbol, also erzeugt \" ein einzelnes Doppelhochkomma und \\ nur einen einzelnen Backslash. Die Worte werden durch Leer- oder Tabulatorzeichen getrennt.

Für jede Zeile gibt es ein Schlüsselwort, das die Bedeutung der Zeile anzeigt und unabhängig von Groß-/Kleinschreibung erkannt wird. TITLE leitet ein neues Menü ein, das rechts vom vorigen Menü erscheinen wird. Sein Argument ist der Text, der als Menü-Überschrift angezeigt werden soll. ITEM ist ein einzelner Menüpunkt mit einem Anzeigetext und einem ARexx-Befehl, der bei Auswahl des Menüpunktes intern aufgestartet wird (siehe ARexx-Kapitel). Zusätzlich ist ein Tastaturkürzel erlaubt (siehe oben). Der Anzeigetext für den Menüpunkt darf beliebig sein (z.B. Deutsch). Falls kein ARexx-Befehl angegeben wird, bereitet ITEM ein Untermenü vor. Die Zeilen des Untermenüs haben SUB statt ITEM als Schlüsselwort. Das Schlüsselwort BAR schließlich setzt eine Trennzeile zwischen zwei Menüpunkte. Es ist nicht möglich, eine Trennzeile in einem Untermenü zu setzen, setzen Sie also nie BAR zwischen die Punkte eines Untermenüs. Der Menüteil wird mit MENUEND beendet.

### 2.10.2 Tastenkonfiguration

Nach MENUEND sind die Tastenkombinationen angegeben, die Befehle auslösen können. Die Kombinationen überlagern die normale Ansteuerung, d.h., Drücken der k-Taste kann anstelle der Ausgabe »k« eine Funktion auslösen. Die Syntax einer Definitionszeile ist (Kommentare wie oben):

```
TastenID ARexxBefehl Qualifier1 Qualifier2 ...
```



Ein Qualifier ist SHIFT, ALT, LAMIGA, CONTROL oder keiner und wird unabhängig von der Groß-/Kleinschreibung erkannt. Der Qualifier entspricht den gleichnamigen Tasten, dabei ist LAMIGA die linke Amiga-Taste (die rechte Amiga-Taste ist den Kürzeln für die Menüs vorbehalten). Es dürfen mehrere bzw. gar keine Qualifier angegeben werden. Der ARexx-Befehl ist derjenige, der bei Erkennen der Tastenkombination ausgelöst wird. TastenID ist entweder ein ASCII-Buchstabe oder ein Mnemonic. Der ASCII-Buchstabe muß in korrekter Kleinschreibung angegeben sein, d.h. nicht in Großbuchstaben, da der Qualifier angibt, ob z.B. gleichzeitig die Shift-Taste verlangt wird. Ein Mnemonic ermöglicht es, Spezialtasten zu erreichen. Es gibt die folgenden Mnemonics, die ebenfalls unabhängig von der Groß-/Kleinschreibung erkannt werden:

```
ESCAPE, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, BACKSPACE, DELETE,
HELP, TABULATOR, RETURN, CURSORUP, CURSORLEFT, CURSORDOWN,
CURSORRIGHT
```

Mnemonics für den seitlichen Zehnerblock sind:

```
PAD[, PAD], PAD/, PAD*, PAD-, PAD+, PAD0, PAD1, PAD2, PAD3, PAD4,
PAD5, PAD6, PAD7, PAD8, PAD9, PAD., ENTER
```

Aus den Namen geht sehr klar hervor, welche Taste jeweils gemeint ist. Nicht definierte Tastenkombinationen bewirken dasselbe, was sie normalerweise auf der eingestellten Tastatur bewirken. Daher sollten einige Einstellungen wie z.B. die folgenden immer in der Setup-Datei enthalten sein:

```
DELETE Delete
BACKSPACE Backspace
RETURN Newline
ENTER Newline
```

Ein und derselben Taste können auch mehrere Kombinationen mit unterschiedlichen Qualifiern diktiert werden:

```
CURSORUP "Cursor Up"
CURSORDOWN "Cursor Down"
CURSORLEFT "Cursor Left"
CURSORRIGHT "Cursor Right"
CURSORUP "Position SOV" SHIFT
CURSORDOWN "Position EOV" SHIFT
CURSORLEFT "Position SOL" SHIFT
CURSORRIGHT "Position EOL" SHIFT
CURSORUP "Previous Page" ALT
CURSORDOWN "Next Page" ALT
CURSORLEFT "Previous Word" CONTROL
CURSORRIGHT "Next Word" CONTROL
CURSORUP "Position SOF" CONTROL SHIFT
CURSORDOWN "Position EOF" CONTROL SHIFT
```

Die Tastenkonfiguration endet mit dem Dateiende der Setup-Datei. Als Abschlußbeispiel soll bei Drücken von <Control>+<n> der nächste Fehler der Fehlerliste angesprungen werden:

```
n NextError CONTROL
```

Eine wichtige Einschränkung gilt, wenn das Pufferlisten-, Marken- oder Findfenster aktiv ist. Dann werden nur solche Tastendrücke ARexx-Befehle auslösen, die von Tasten kommen, welche keine ASCII-Buchstaben bzw. -zeichen erzeugen, d.h. nur von Funktionstasten, Help-Taste, Cursor-Tasten etc. Das obige <Control>+<n> würde also verschluckt, wenn das Pufferlistenfenster gerade aktiv ist.

### 2.10.3 Ausgeschaltete Menüs und Checkmarks

Der Editor ist in der Lage, bei ARexx-Befehlen auf Menüpunkten die Punkte auszuschalten (*ghosting*) oder mit einem Checkmark zu versehen. Da dies einen hohen internen Aufwand verursacht, ist dies nur für einige wichtige Befehle implementiert, ferner sollte bei dem Befehl SETPREFS darauf geachtet werden, daß nach dem Befehl genau ein Leerzeichen folgt und darauf gleich der Name des zu beeinflußenden Parameters, ansonsten wird der Menüpunkt nicht richtig mit einem Checkmark versehen (siehe Standardkonfiguration). Korrekt ist z.B.

```
ITEM Overstrike "SetPrefs Overstrike Toggle"
```

# Kapitel 3

# Optimierender Makro-Assembler V2.1

## 3.1 Einführung

Dieses Assembler-Handbuch dient nicht als Einführung in die Programmierung der Prozessoren der Motorola-68000-Familie. Dafür gibt es im Buchhandel sehr gelungene Werke verschiedener Autoren, auf die ich hier verweisen möchte. Es wird vorausgesetzt, daß Sie die gängige Terminologie für 68000-Assembler kennen.

Nach diesem Hinweis möchte ich Sie beim Optimierenden-Makro-Assembler herzlich willkommen heißen. OMA ist einer der leistungsfähigsten, für die Commdore-Amiga-Familie erhältliche Makro-Assembler. Neben seiner hohen Geschwindigkeit besitzt OMA Kompatibilität zum HiSoft-Devpac-Assembler V2.0 sowie zum Metacomco-Makro-Assembler V11.00, der den ersten Standard für Assembler auf dem Amiga gesetzt hat. Genauere Unterschiede finden Sie im Kapitel 3.7.

OMA ist als Nachfolger des Metacomco-Assemblers gedacht und unterstützt den Motorola-Standard für 68000-Assembler. Zum Konzept gehört die möglichst uneingeschränkte Benutzung aller Funktionen, so daß z.B. Makros oder Includefiles beliebig tief geschachtelt werden können. Die vielen präzisen Fehlermeldungen helfen, Quelltexte schnell zu korrigieren, wobei Sie besonderen Komfort genießen, wenn Sie den Assembler in den Editor integrieren (dazu sehen Sie bitte im Editor-Handbuch nach).

OMA erzeugt Code für alle Mitglieder der Motorola-68000-Familie, auch für PMMUs und FPUs. Ich erwähne im folgenden ausdrücklich, wo ein Unterschied zwischen den einzelnen Prozessoren besteht. OMA verarbeitet als Quelltexte normale ASCII-Dateien, wie sie von üblichen Editoren erzeugt werden, und erzeugt Objektfiles oder direkt ausführbare Programme im Amiga-DOS-Hunkformat. Durch diese Standardisierung ist es möglich, den zum OMA-Paket gehörenden HiTex-Editor oder den Linker DLink durch Ihren Lieblings-Editor oder -Linker zu ersetzen. Obwohl dabei die Möglichkeit

verlorengeht, Editor und Assembler zu einem integrierten Paket zusammenzusetzen, gibt dies Ihnen größere Flexibilität im Umgang mit dem Paket.

Hier noch eine kleine Zusammenfassung der Leistungsmerkmale des Assemblers:

- kennt 68000, 68010, 68020, 68030, 68040, 68881, 68882, 68851
- unterstützt Source-Level-Debugging
- unterstützt objektorientierte Programmierung, siehe 7
- kompatibel zum Devpac- und Metacomco-Assembler, Zusatzdirektiven
- kommandozeilen-kompatibel zum Metacomco, Zusatzoptionen
- läuft auf allen Amiga-Betriebssystemversionen
- ab Workbench 1.3 resident haltbar
- über 300 Fehlermeldungen und Warnungen
- Listingerzeugung und -formatierung wie Metacomco, Zusatzdirektiven
- Direktiven sind redefinierbar, daher mögliche Cross-Assemblierung für andere Prozessoren
- beliebig viele Include-Directories, ein Default-Include-Directory
- Zeilenlänge bis zu 255 Zeichen, freies Zeilenformat
- Symbollänge bis zu 255 Zeichen, beliebig viele Symbole
- beliebige Makrogröße und -verschachtelung
- 36 Makroparameter plus Null-Parameter, Weiterzähl-Parameter und Spezial-Makrobefehle
- beliebige Includefile-Verschachtelung
- beliebige Verschachtelung von IF-Direktiven
- bis zu 255 Sektionen
- Ausdruckschachtelung bis zu 32 Klammern
- stellenweise oder global ein- und ausschaltbarer Optimierer erzeugt hochoptimierten Code
- hochoptimierender Mehrpass-Modus
- hohe Assembliergeschwindigkeit: ca. 63000-74000 Zeilen pro Minute, Peak- Geschwindigkeit von 200000 Zeilen pro Minute (NOPs)
- wenn in den Editor integriert: direktes Assemblieren aus den Editorpuffern, direktes Fehleranspringen und Assemblieren in den Speicher
- kann direkt ausführbare Programme erzeugen
- vorassemblierte Includefiles möglich
- »Small data«-Modell (siehe Kapitel 3.11)
- geringer Speicherbedarf



Zum Abschluß dieses Kapitels möchte ich noch ein paar terminologische Erklärungen machen:

Ich verwende »PC« für den Program Counter des Prozessors, während »Programmzähler« sich auf den internen Zähler des Assemblers bezieht, der bestimmt, wo der erzeugte Code im Programm zu liegen kommt. Ein Begriff, der in eckige Klammern eingefaßt ist, ist optional, kann also weggelassen werden oder darf höchstens einmal angegeben sein, z.B. »[Label]«.

In allen Fällen, in denen der Assembler für Direktiven Strings als Operanden liest, darf der String auch in einfache oder doppelte Hochkommata eingefaßt sein und kann dann Leerzeichen enthalten. Soll dann das einfassende, einfache bzw. doppelte Hochkomma selber in dem String erscheinen, muß es zweimal angegeben werden, z.B. 'Was"n das' oder "Was""n das". Diese Regeln möchte ich als übliche Regeln für Assemblerstrings vereinbaren. Eine Ausnahme hiervon bilden Symbolstrings.

Den Begriff »Case-Unterschied« verwende ich aus verständlichen Gründen an solchen Stellen, an denen eigentlich »Unterschied zwischen Groß- und Kleinschreibung« stehen sollte. »Groß- und Kleinschreibung« bezieht sich hier aber auf alle Buchstaben eines Strings, nicht nur auf den ersten!

Für weitergreifende Beispiele in der Programmierung des Assemblers verweise ich auf das Examples-Directory der Originaldiskette. Dort finden Sie Beispiele für IF-Verschachtelungen, Makros usw. in Hülle und Fülle.

Es soll nicht unerwähnt bleiben, daß der Assembler seine Tauglichkeit im Alltagsbetrieb schon bewiesen hat: Das komplette Assemblerpaket wurde mit OMA selber geschrieben!

## 3.2 Aufruf des Assemblers

### 3.2.1 Aufruf

Der Assembler kann nur über einen CLI-Aufruf (und damit auch z.B. durch das Make-Utility des Editors) oder über den Assemble-Requester des Editors gestartet werden. Er versteht Dateinamen in der CLI-Syntax, so daß der Parameterstring »a**b ist c« den Namen »a*b ist c« ergibt. Wenn Quellfile der Name des zu assemblierenden Files ist, lautet die Aufrufsyntax:

```
OMA [PROG|FROM] Quellfile [Hauptoptionen]
```

*PROG* oder *FROM* darf optional dem Quellfile-Namen vorangestellt werden. Wenn der Assembler das Quellfile nicht öffnen kann und dieser Name nicht mit ».asm« endet, versucht er, das File mit dem Suffix ».asm« zu öffnen, d.h. z.B. Quellfile.asm. Dieses Suffix ist die *Standardendung* für Assemblerquelltexte. Beachten Sie, daß bei Weglassen des Suffix evtl. ein anderes File als der von Ihnen gewünschte Quelltext assembliert wird!

*Hauptoptionen* dürfen weggelassen werden. Für sie gibt es die folgenden Möglichkeiten:

`-o Objektfile`

Mit dieser Option erhält das Objektfile den Namen Objektfile. Fehlt sie, dann erhält das Objektfile einen Namen, der sich aus dem Quellfile-Namen ohne evtl. vorhandene ».asm«-Endung und dem Suffix ».obj« ergibt. Ist das Objektfile jedoch ein ausführbares Programm, entfällt das ».obj«-Suffix. Siehe auch OUTPUT-Direktive.

`-l Listingfile`

Mit dieser Option erzeugt der Assembler ein Listingfile namens Listingfile. Diese Option kann nicht zusammen mit der »-c l«-Option benutzt werden, und der Fast-Modus wird ausgeschaltet (siehe Option »-n«). Der Assembler erzeugt ein Listingfile nur auf Befehl. Siehe auch Kapitel 3.12.

`-v Meldungsfile`

Mit dieser Option leiten Sie die Standardmeldungen des Assemblers vom Terminal (dem aktuellen CLI-Fenster) in das File Meldungsfile um. Beachten Sie, daß dies nicht das Listingfile beeinflußt und daß dadurch während der Assemblierung keine Ausgabe in das CLI-Fenster erfolgt.

`-h Headerfile`

Mit dieser Option befehlen Sie dem Assembler, vor jedem Pass zusätzlich das Includefile Headerfile zu inkludieren. Dies kann nützlich sein, um Makros einzubeziehen, die für die Kompatibilität zu anderen Assemblern zuständig sind. Siehe Kapitel 3.7 und die INCLUDE-Direktive.

`-e Equatefile`



Mit dieser Option schreiben Sie alle Quelltextzeilen des Assemblerdurchlaufes, die Zuweisungen wie z.B. EQU, EQUR, RS, OFFSET und dessen DS-Zuweisungen enthalten, in das File Equatefile. Dadurch erhalten Sie eine Zusammenfassung aller Zuweisungszeilen des assemblierten Quelltextes.

`-t Tabellenfile`

Mit dieser Option werden alle absoluten Symbole und Makros des Assemblerlaufes in einem speziellen Format in das File Tabellenfile geschrieben. Sie können damit Includefiles vorassemblieren und so Zeit für einen Assemblerlauf sparen. Siehe Kapitel 3.13.

`-p Tabellenfile`

Mit dieser Option wird das durch die »-t«-Option erzeugte File Tabellenfile vor dem Assemblerlauf eingelesen, und seine Symbole und Makros werden in die Assemblierung einbezogen. Ein vorassembliertes File kann auch mit der INCPRE-Direktive gelesen werden. Siehe Kapitel 3.13.

`-s`

Der Assembler gibt am Ende der Assemblierung eine Statistik über den Assemblerlauf aus. Diese enthält die Größe der insgesamt erzeugten Sektionen, eine Angabe über den Speicherbedarf, die Laufzeit und die Anzahl der insgesamt sowie die Anzahl der pro Minute assemblierten Zeilen.

`-m`

Schaltet den Mehrpass-Modus ein. Siehe Abschnitt 3.2.2 unten.

`-n`

Der Fast-Modus wird ausgeschaltet. Siehe Abschnitt 3.2.2 unten.

`-d Symbol[:Wert]`

Diese Option definiert das absolute Symbol *Symbol* für den Assemblerlauf. Wenn der optionale Hexadezimalwert *Wert* (durch einen Doppelpunkt von *Symbol* getrennt) angegeben wird, bekommt *Symbol* diesen Wert statt normalerweise 0. *Wert* darf nicht den Hexadezimal-Vorsatz »$« haben. Diese Option ermöglicht Ihnen, durch Steuerung von außen und IF-Direktiven Quelltext-Abschnitte vom Assemblerlauf auszuschließen oder einzubeziehen.

`-i Includedirectory`

Sie können mit dieser Option Directories zum Suchen von Includefiles angeben (siehe INCLUDE-Direktive). Es sind mehrere Directories möglich, wenn Sie die »-i«-Option mehrfach anwenden oder nur eine »-i«-Option verwenden und die einzelnen Directorynamen durch ein Pluszeichen oder Komma trennen, z.B. »-i a+b« oder »-i a,b«. Soll ein Directoryname ein Pluszeichen, Leerzeichen oder Komma enthalten, muß er durch eine einzelne »-i«-Option angegeben werden und in Hochkommata eingefaßt sein (CLI-Konvention). Sie dürfen beliebig viele Include-Directories angeben. Wenn in Ihrem System ein Directory namens AINC: existiert, erkennt der Assembler dies automatisch und

benutzt es, als ob es als letztes Include-Directory mit der »-i«-Option angegeben wäre. Sie können also die Angabe eines Standard-Includedirectories vermeiden, wenn Sie z.B. `assign ainc: df0:include` unter dem CLI ausführen lassen (evtl. in der Startup-Sequenz). Obwohl Includefiles gelesen werden, lautet der Assembler-Aufruf dann z.B. nur noch `OMA m.asm`. Das Standard-Directory (Default-Includedirectory) AINC: wird erst nach allen anderen Directories durchsucht.

`-c Nebenoptionen`

Der Assembler schaltet Nebenoptionen ein, die ungetrennt direkt hintereinander stehen müssen. Für die »-c«-Option existiert das Synonym »OPT Nebenoptionen«.

Gültige Nebenoptionen für »-c« sind:

**c** Es gibt keinen Case-Unterschied der Symbole. Alle Symbole gelten als großgeschrieben und gelangen auch so in das Objektfile. Das bedeutet u.U., daß Sie beim Linken mit DLink die NOCASE-Startoption angeben müssen.

**d** Lokale Label werden nicht ins Objektfile (in den Hunk_Symbol) gelistet.

**k** Das Listing wird vorerst in IF-Direktivenabschnitten mit Falsch- Resultaten unterdrückt. Diese Option kann durch die CLIST- und NOCLIST-Direktiven unwirksam gemacht werden (siehe auch dort).

**l** Der Assembler erzeugt ein Listingfile, dessen Name sich aus dem Quellfilenamen ohne evtl. vorhandene ».asm«-Endung und dem Suffix ».lst« ergibt. Der Fast-Modus wird ausgeschaltet. Diese Option kann nicht zusammen mit der »-l«-Option benutzt werden.

**m** Das Listing wird während der Makroexpansion unterdrückt. Diese Option kann durch die MLIST- und NOMLIST-Direktiven unwirksam gemacht werden (siehe auch dort).

**n** Die Produktion eines Objektfiles wird verhindert. Sie können dies zum schnellen Test von Quelltexten verwenden. Siehe auch NOOBJ-Direktive.

**o** Der Optimierer wird vorerst abgeschaltet. Diese Option kann durch die OPTIMIZE- und NOOPTIM-Direktiven bzw. durch »OPT o« unwirksam gemacht werden (siehe auch dort).

**p** Es wird vorerst kein Seitenvorschub im Listing erzeugt. Diese Option kann durch die PAGE- und NOPAGE-Direktiven unwirksam gemacht werden (siehe auch dort).

**q** Mit dieser Option werden zum Source-Level-Debuggen Quelltextzeilen in einem speziellen Format in einen Debughunk abgespeichert. Siehe auch Kapitel 3.14 über Source-Level-Debugging.

**s** Der Assembler schreibt die relativen Symbole ins Objektfile in einen Symbolhunk. Solch ein Programm kann dann in Verbindung mit einem symbolischen Debugger leicht lesbar gemacht werden. Zur Entfernung der Symboltabellen aus Programmen, Objektfiles und Libraries sehen Sie sich bitte das Hilfsprogramm *Strip* an.



**t** Nach dem Assemblerlauf wird eine Symboltabelle in das Listingfile gegeben. Siehe Kapitel 3.12.

**w** Sogenannte *Warnings*, d.h. Warnungen des Assemblers, die auf merkwürdige Zustände hinweisen und keine Fehler sind, werden unterdrückt. Siehe Kapitel 3.10.

**x** Nach dem Assemblerlauf wird eine Cross-Referenz in das Listingfile gegeben. Siehe Kapitel 3.12.

**z** Diese Option wählt das Assemblieren in den Speicher an und hat nur eine Auswirkung, wenn der Assembler unter dem Editor resident ist. Sie werden diese Option benutzen, wenn Sie mit dem Editor-Make-Utility assemblieren und das erzeugte Programm direkt unter dem Editor ausführen lassen möchten. Beachten Sie, daß diese Option die Objektfile-Produktion nicht beeinflußt.

Es gibt keinen Case-Unterschied für die Options-Buchstaben, weder für Haupt- noch für Nebenoptionen. Soll der Quellfilename die Form »-?« haben (»?« steht für irgendein Zeichen), muß er in Hochkommata eingefaßt sein, damit er nicht für eine Option gehalten wird.

Der Assembler erzeugt automatisch direkt ausführbare Programme, wenn im Quelltext weder XREF- noch XDEF-Direktiven vorkommen (ein völlig abgekapseltes Modul wäre schwer zu linken, womit auch?), sonst nur linkbare Objektfiles, da Referenzen nicht aufgelöst sind. Es ist jedoch möglich, mit der Direktive »OPT L+« oder »OPT L-« zwingend zwischen linkbar und ausführbar zu wählen.

### 3.2.2 Fast-Modus und Mehrpass-Modus

Normalerweise arbeitet der Assembler immer im Fast-Modus. Dieser schnelle Modus wird nur durch die »-n«-Option, das Erzeugen eines Listingfiles oder das Auftreten von Fehlern im ersten Pass abgeschaltet. Der Fast-Modus ist schnell, da der Quelltext im zweiten Pass nicht mehr gelesen werden muß, schluckt aber viel Speicher. Deswegen können Sie mit der »-n«-Option Platz sparen und sehr große Quelltexte assemblieren, verlieren dann aber an Geschwindigkeit. Bitte beachten Sie, daß Quellzeilen bei Fehlermeldungen im zweiten Pass des Fast-Modus leicht umformatiert ausgegeben werden.

Der Mehrpass-Modus wurde geschaffen, damit Sie leicht hochoptimierten Code erzeugen können. Sie schalten ihn mit der »-m«-Option ein, und es werden so lange neue Quelltext-Durchläufe (Passes) eingeschoben, bis der Assembler den Code nicht mehr stärker optimieren kann. Diesen Modus sollten Sie verwenden, wenn Ihr Programm fertig ist, da er prinzipbedingt sehr langsam arbeitet.

### 3.2.3 Beispielaufrufe

```
OMA prog.asm
OMA prog
```

Assembliert prog.asm als Objektfile prog.obj bei Benutzung des Fast-Modus. Da der Assembler automatisch die ≫.asm≪-Endung anhängt, darf beim zweiten Beispiel kein File namens prog existieren, damit prog.asm assembliert wird.

```
OMA prog.asm -o Prog -l * -i include1 -i include2 OPT pwt -s
OMA prog.asm -o Prog -l * -i include1+include2 OPT pwt -s
OMA prog.asm -o Prog -l * -i include1,include2 OPT pwt -s
```

Assembliert prog.asm als Objektfile Prog, schreibt das Listing zum Terminal, benutzt die Includedirectories include1 und include2, unterdrückt den Listing-Seitenvorschub und die Warnings und gibt am Ende die Symboltabelle ins Listing sowie eine Statistik über den Assemblerlauf zum Terminal.

```
OMA prog.asm -m -c s -d DEBUG
```

Assembliert prog.asm als Objektfile prog.obj im hochoptimierenden Mehrpass-Modus, schreibt die Symboltabelle ins Objektfile und definiert das absolute Symbol *DEBUG* für den Assemblerlauf (mit dem Wert 0). Im Quelltext können Abschnitte per IFD- und IFND-Direktive ein- oder ausgeschlossen werden.

### 3.2.4 Anhalten des Assemblers

Sie können den Assembliervorgang jederzeit mit <Ctrl>+<c> anhalten.



## 3.3 Die Zeilensyntax

### 3.3.1 Die Syntax

Der Assembler verarbeitet Zeilen aus normalen ASCII-Dateien. Ein Zeilenende liegt vor, wenn das ASCII-Zeichen $0A (Linefeed), das ASCII-Zeichen $0D (Carriage Return) oder die Folge $0D $0A (Carriage Return - Linefeed) gefunden wird. Der zum OMA-Paket gehörende HiTex-Editor erzeugt als Zeilenende ein einzelnes Linefeed. Eine Zeile darf maximal 255 Zeichen lang sein, die restlichen Zeichen werden nicht mehr berücksichtigt, sondern überlesen, und es wird ein Warning erzeugt.

Eine Kommentarzeile ist eine leere Zeile oder eine mit einem Stern ≫*≪ als erstem Zeichen. Kommentarzeilen werden ignoriert und lösen keine Aktion aus.

Alle anderen Zeilen dürfen höchstens eine Assembler-Direktive enthalten und müssen der folgenden Syntax gehorchen:

```
[Label][:] [Opcode][.Size] [Operanden] [Kommentar]
```

*Label* darf bei den meisten Direktiven weggelassen werden und optional mit einem Doppelpunkt enden. Der Doppelpunkt muß angegeben werden, wenn *Label* nicht auf dem ersten Zeichen der Zeile beginnt. Label ist der Ausdruck für dieses Feld; wenn eine Variable, Konstante oder ein Makro selber gemeint ist, wird der Ausdruck *Symbol* verwandt. Eine Größenangabe wie *Size* ist am Ende vom Label nicht zulässig. Ein normales Label gehorcht den gleichen Bedingungen wie der Opcode (siehe folgend), aber beachten Sie unbedingt, daß lokale Label bzw. Symbole eine andere Syntax haben (siehe unten).

*Opcode* ist eine Prozessorinstruktion, eine Assemblerdirektive oder ein Makroname. Der Opcode muß mit einem Buchstaben, Punkt oder Unterstrich beginnen und darf danach zusätzlich eine Dezimalziffer enthalten. Als Buchstaben sind auch internationale Zeichen wie z.B. ≫ö≪ oder ≫æ≪ zulässig. Siehe auch Kapitel 3.6.

*Size* ist eine optionale Größenangabe für den Opcode und muß, falls angegeben, durch genau einen Punkt getrennt direkt hinter dem Opcode stehen. Eine Größenangabe ist für manche Opcodes nicht zulässig.

*Operanden* gibt es nur, wenn der Opcode solche besitzen kann, sonst gelten sie als Kommentar. Mehrere Operanden werden durch Kommata getrennt und dürfen keine Leer- oder Tabulatorzeichen enthalten. Ein Ausnahme hiervon bilden Parameter beim Makroaufruf, dort darf ein in spitze oder eckige Klammern eingefaßter Parameter Leer- oder Tabulatorzeichen enthalten.

*Kommentar* ist optional und muß mit einem Semikolon beginnen, sofern es keinen Opcode gibt. Sonst ist es egal, mit welchem Zeichen der Kommentar beginnt. Sie sollten ihn aber wegen Quelltextkompatibilität zu anderen Assemblern immer mit einem Semikolon einleiten.

Die einzelnen Felder müssen mit mindestens einem Leer- oder Tabulatorzeichen getrennt sein. Die einzige Ausnahme ist der Opcode ≫=≪ als Synonym für EQU, vor und hinter dem Label und Operand direkt stehen dürfen. Die Syntax entspricht der des Devpac- und Metacomco-Assemblers und bietet beliebig formatierbare Quelltexte.

### 3.3.2 Lokale Symbole

Ein lokales Symbol ist nur zwischen der Definition von zwei »echten«, relativen Symbolen gültig. Es gilt als lokal, wenn es mit einer Dezimalzahl beginnt und als letztes Zeichen das Dollarzeichen »$« hat. Beispiel:

```
start nop
      nop
1$    move.b (a0)+,(a1)+  ; lokale Definition
      bne.s  1$           ; bezieht sich auf das vorige "1$"
ende
      bra.s  1$           ; bezieht sich NICHT auf das obige "1$"
```

Ein Symbol gilt ebenfalls als lokal, wenn es mit einem Punkt beginnt:

```
start nop
      nop
.lab  move.b (a0)+,(a1)+
      bne.s  .lab
ende
      bra.s  .lab         ; bezieht sich NICHT auf das obige ".lab"
```

Sie können das Einleitungszeichen Punkt in den Unterstrich »_« ändern, indem Sie die Direktive »OPT U+« verwenden (siehe auch dort):

```
      opt u+
start nop
      nop
_lab  move.b (a0)+,(a1)+
      bne.s  _lab
ende
      bra.s  _lab         ; bezieht sich NICHT auf das obige "_lab"
```

Sollten Sie sowohl den Punkt als auch den Unterstrich für einen normalen Symbolnamen benötigen, dann können Sie durch die Direktive »OPT U-« bewirken, daß nur noch die Zahlenlabel der Form »xxx$« als lokale Symbole gültig sind.



## 3.4 Ausdrücke

### 3.4.1 Symbolsyntax und Konstanten

Ein nicht lokales Symbol (siehe oben) muß mit einem Buchstaben, Punkt oder Unterstrich beginnen. Nach dem ersten Zeichen darf zusätzlich eine Dezimalziffer im Symbolnamen vorkommen. Es besteht der Case-Unterschied, solange nicht die »-c c«-Option beim Aufruf benutzt wird. Als Buchstaben sind im Symbolnamen auch internationale Zeichen wie z.B. »ö« oder »æ« zulässig, so daß der Symbolname »Ärger« korrekt ist.

Ein Symbol ist absolut, wenn es einem absoluten Ausdruck zugewiesen wird, z.B.:

```
AbsolutesSymbol EQU 5*3
```

Ein Symbol ist relativ, wenn es einem relativen Ausdruck, d.h. dem aktuellen Programmzähler, zugewiesen wird. Lokale Symbole sind immer relativ. Beispiele:

```
Relativ  EQU  *      ; Zuweisung an den aktuellen Programmzähler
1$       nop
rela     nop
nochwas  equ   rela  ; Zuweisung an relatives Symbol
```

Ein Symbol gilt als importiert, wenn es durch die XREF-Direktive definiert wurde. Für importierte Symbole sind nur wenige Rechenoperationen möglich.

Sie müssen also absolute, relative oder importierte Symbole unterscheiden. Durch die unten folgenden Rechenregeln überträgt sich diese Eigenschaft auch auf Teilausdrücke. Konstanten gelten immer als absolut; es gibt die folgenden Konstanten:

1. Eine Dezimalzahl besteht nur aus Dezimalziffern, z.B. »1234«.
2. Eine Hexadezimalzahl beginnt mit dem Dollarzeichen »$«, und es folgen hexadezimale Ziffern »0« bis »9« sowie die Buchstaben »a« bis »f«, z.B. »$1a3F« für den Dezimalwert 6719. Es gibt keinen Case-Unterschied.
3. Eine Binärzahl besteht aus einem Prozentzeichen, gefolgt von binären Ziffern »0« oder »1«, z.B. »%01001« für den Dezimalwert 9.
4. Eine Oktalzahl beginnt mit dem kommerziellen »At«-Zeichen »@«, und es folgen oktale Ziffern »0« bis »7«, z.B. »@2347« für den Dezimalwert 1255.
5. Eine Stringkonstante beginnt mit einem einfachen oder doppelten Hochkomma und endet dann auch mit dem einfachen oder doppelten Hochkomma (nicht mit dem jeweils anderen!). Sie darf maximal vier Zeichen lang sein und wird rechtsseitig aufgefüllt. Der Case-Unterschied besteht hier. Soll ein einfaches bzw. doppeltes Hochkomma selber als Zeichen vorkommen, müssen Sie es zweimal angeben. Dies ist im übrigen für alle Fälle gültig, in denen der Assembler einen String liest, z.B. auch für INCLUDE, DC.B, IDNT, TTL usw. Beispiele:

```
'2-3'          Wert: $00322D33, rechtsseitiges Auffüllen
"2-3"          Wert: $00322D33, rechtsseitiges Auffüllen
```

```
''          Wert: $00000000
'It''s'     Wert: $49742773
"It's"      Wert: $49742773
"It""s"     Wert: $49742773
"It""s'     Fehler, hätte mit doppeltem Hochkomma enden müssen
```

### 3.4.2 Rechenregeln

Der Assembler wertet Ausdrücke nach hierarchischem Prinzip aus, so daß z.B. Punkt- vor Strichrechnung geht. Ein Teilausdruck oder Operand darf ein Symbol, der aktuelle Programmzähler »*« oder eine Konstante sein. Beachten Sie, daß der Stern »*« im Offsetabschnitt (siehe OFFSET-Direktive) nicht den *relativen* Programmzähler, sondern den *absoluten* Offsetzähler repräsentiert, und daß bei ganzzahligen Ausdrücken keine Fließkommasymbole erlaubt sind (siehe auch unten). Bei ganzzahligen Ausdrücken wird immer mit ganzzahligen, vorzeichenbehafteten Werten von 32 Bit Länge gerechnet. Folgende Operatoren sind gültig:

| *Priorität* | *Operator* | *Bedeutung* |
|---|---|---|
| 5 | + | Monadisches Plus |
| 5 | - | Monadisches Minus |
| 5 | ~ | Logisches Negat (Tildezeichen) |
| 4 | >> | Rechtsverschieben (zweimal ">") |
| 4 | << | Linksverschieben (zweimal "<") |
| 3 | ^ | Logisches Exklusiv-Oder ("Dach") |
| 3 | & | Logisches Und |
| 3 | \| | Logisches Oder |
| 3 | ! | Logisches Oder |
| 2 | * | Multiplikation |
| 2 | / | Division |
| 2 | % | Modulo-Operator |
| 1 | + | Addition |
| 1 | - | Subtraktion |

Operatoren mit höherer Priorität werden zuerst berechnet. Die Priorität kann durch Klammerung durchbrochen werden, wobei die Klammerverschachtelung zusammen mit der Operatorverschachtelung bis zu 32 Ebenen tief reicht. Die Operatoren mit der Priorität fünf sind monadische Operatoren, haben also nur einen Operanden, der unmittelbar auf den Operator folgt.

Alle Operatoren mit Priorität größer eins können nur mit absoluten Werten oder Symbolen als Operanden arbeiten. Das Ergebnis ist immer ein absoluter Wert. Das monadische Plus akzeptiert als einzige Ausnahme alle möglichen Operanden.

Für die Addition sind folgende Kombinationen von absoluten, relativen und importierten Operanden gültig (z.B. ergibt absoluter Operand plus relativer Operand einen relativen Wert bzw. evtl. neuen Operanden):



| *Addition* | absolut | relativ | importiert |
|---|---|---|---|
| absolut | absolut | relativ | importiert |
| relativ | relativ | FEHLER | FEHLER |
| importiert | importiert | FEHLER | FEHLER |

Für die Subtraktion sind folgende Kombinationen gültig (z.B. ergibt relativ minus relativ einen absoluten Wert oder absolut minus relativ einen Fehler):

| *Subtraktion* | absolut | relativ | importiert |
|---|---|---|---|
| absolut | absolut | FEHLER | FEHLER |
| relativ | relativ | absolut | FEHLER |
| importiert | importiert | FEHLER | FEHLER |

Nun noch ein paar Beispielausdrücke:

```
3+4*5           ergibt 23 absolut
'2'+start       ergibt $34 relativ, wenn ">start"< den Wert 2 relativ hat
"2'             ergibt einen Fehler, korrekt ist "2" oder '2'
*-start         ergibt 4 absolut, wenn ">start"< den Wert 2 relativ hat
                und der Programmzähler (immer relativ) den Wert 6
~1              ergibt $FFFFFFFE = -2
(8>>2)          Acht, zweimal nach rechts geshiftet (ergibt 2 absolut)
(1<<7)          Eins, siebenmal nach links geshiftet (-> $80 absolut)
(1<<7)!1        ergibt $81 absolut
(80+128)<<16!(32+128)<<4      ergibt $00D00A00 absolut
```

### 3.4.3 Fließkomma-Ausdrücke

Ausdrücke bei der FEQU- oder FSET-Direktive oder bei FPU-Befehlen mit der Größenangabe S, D, X oder P sind Fließkomma-Ausdrücke, die immer als absolut geltende Fließkommasymbole enthalten dürfen. Ganzzahlige Symbole werden im Fließkomma-Ausdruck erst in Fließkommawerte umgewandelt; zudem sind nur absolute Werte erlaubt. Beachten Sie jedoch, daß solche Fließkomma-Ausdrücke nur verarbeitet werden können, wenn ihr Rechner eine **FPU besitzt**. Gerechnet wird dann mit Hilfe der FPU mit Extended-Precision-Zahlen mit 19 bis 20 Stellen Genauigkeit im Bereich $6*10^{4931}$ bis $9*10^{-4952}$.

Als Operatoren sind nur noch Addition, Subtraktion, Multiplikation, Division und der Modulo-Operator erlaubt, wobei die obige Priorität weiterhin gilt. Ferner sind Fließkommazahlen erlaubt, welche aus einem Vorzeichen, einer Dezimalziffer oder einem Punkt und darauf folgenden Nachkommastellen, sowie einem evtl. ebenfalls vorzeichenbehafteten Exponenten bestehen. Leerzeichen dürfen zwischendurch nicht vorkommen.

Beispiele für Fließkomma-Zahlen und -Ausdrücke:

```
0.0001
0.0001e3   ist   0.1
3.141592653589793
-3E-10
-.4E20   ist   -4E19
-.4E+20
+78
Pi*3+Wert   wobei Pi Fließkomma- und Wert Integersymbol ist
```



## 3.5 Adressierungsarten

OMA unterstützt alle gängigen Adressierungsarten der Motorola-Prozessoren wie von Motorola definiert. Bezüglich Optimierungen sei auf das Kapitel 3.9 verwiesen. Im folgenden bezeichnet ≫a≪ einen absoluten Ausdruck, ≫r≪ einen relativen und ≫d≪ einen Ausdruck, der absolut oder relativ sein kann. Für die Register des Prozessors gibt es keinen Case-Unterschied, d.h., es gilt

`sp = Sp = sP = SP` und `D0 = d0`

### 3.5.1 Normale Adressierungsarten

**Dn**

Datenregister direkt. Für die Datenregister kann per EQUR und SETR ein Registersymbol definiert werden. Beispiele: "d0" "D5"

**An**

Adressregister direkt. Für die Adreßregister kann per EQUR und SETR ein Registersymbol definiert werden. Beispiele: "A5" "a0"

**(An)**

Adressregister indirekt. Beispiele: "(a3)" "(A0)"

**(An)+**

Adreßregister indirekt mit Postinkrement. Beispiele: "(a3)+" "(A0)+"

**-(An)**

Adressregister indirekt mit Predekrement. Beispiele: "-(a3)" "-(A0)"

**a(An)**

Adressregister indirekt mit vorzeichenbehaftetem 16-Bit-Offset. Wenn der Offset Null ist, kann zu ≫(An)≪ optimiert werden. Beispiele: "5(a3)" "5(A0)"

**a(An,Xn.x)**

Adressregister indirekt mit vorzeichenbehaftetem 8-Bit-Offset und Index. Der Index darf ein Daten- oder Adreßregister mit der optionalen Größenangabe ≫.w≪ oder ≫.l≪ sein. Beispiele: "5(a3,d0)" "5(a3,d0.w)" "5(a1,a0)" "5(a0,a5.W)" "1(a6,d0.l)" "-45(a4,a2.L)"

**(An,Xn.x)**

Diese Schreibweise für ≫a(An,Xn)≪ ist erlaubt, wenn der Offset Null sein soll.

**a**

Absolut lang. Diese Adressierungsart kann zu ≫Absolut kurz≪ optimiert werden. Beispiele: "wert" "5" (würde zu ≫Absolut kurz≪ optimiert)

**a.L**

Absolut lang. Diese Schreibweise verhindert immer, daß zu »Absolut kurz« optimiert wird. Beachten Sie, daß Sie den Ausdruck nicht in Klammern einfassen müssen. Beispiele: "5.l" "5.L" "wert.l" "wert.L"

**a.W**

Absolut kurz. Der Wert »a« muß im Bereich -32768 bis 32767 liegen. Beispiele: "5.w" "4.W"

**r**

Absolut lang mit 32-Bit-Relokation. Diese Adressierungsart wird evtl. zu »PC-relativ« optimiert. Beispiele: "start" "*"

**r.L**

Absolut lang mit 32-Bit-Relokation. Diese Schreibweise verhindert in jedem Fall, daß zu »PC-relativ« optimiert wird. Beispiele: "start.l" "start.L"

**r.W**

Absolut kurz mit 16-Bit-Relokation. Es wird ein Warning erzeugt, da 16-Bit-Relokationen auf dem Amiga nur in den seltensten Fällen zu lauffähigen Programmen führen. Sie sollten dies nur unter ganz verrückten Umständen benötigen (z.B. bei ROM-Programmen).

**r(PC)**

PC-relativ mit vorzeichenbehaftetem 16-Bit-Offset. Beachten Sie, daß PC-relative Adressierung nur innerhalb derselben Sektion erlaubt ist (siehe SECTION-Direktive). Beispiele: "start(PC)" "ende(pc)"

**r(PC,Xn.x)**

PC-relativ mit vorzeichenbehaftetem 8-Bit-Offset und Index. Der Index darf ein Daten- oder Adreßregister mit der optionalen Größenangabe ».w« oder ».l« sein. Beispiele: "start(PC,d0)" "ende(pc,d0.L)"

**r(Xn.x)**

Diese verkürzte Schreibweise von »r(PC,Xn)« wird erkannt, wenn der Ausdruck relativ ist, und kann leicht mit »d(An)« verwechselt werden. Daher generiert OMA bei jeder Benutzung ein Warning, das Sie auffordert, die Schreibweise mit »PC« zu verwenden. Beispiele: "start(d0)" "ende(d0.L)"

**#a**

Unmittelbar absolut. Der Prozessorbefehl entscheidet, ob ein 8-, 16- oder 32-Bit-Wert generiert wird oder ob der unmittelbare Wert in die Prozessorinstruktion eingebettet wird. Beispiele: "#5" "#wert"

**#r**

Unmittelbar mit Relokation. Der Prozessorbefehl bestimmt, ob ein 8-, 16- oder 32-Bit-Wert generiert wird. Beispiele: "#*" "#start"



### 3.5.2 Adressierungen ab MC68020

Bitte beachten Sie, daß die Adressierungsarten ab MC68020 erst erreichbar sind, wenn Sie per MC68020-Direktive oder einer ähnlichen einen Prozessor ab MC68020 ausgewählt haben.

Bei Indexregistern »Xn« gibt es nun auch das Scaling, bei dem der Inhalt es Indexregisters vor Verwendung mit 1, 2, 4 oder 8 multipliziert wird, z.B. »D5.W*4«. Zur Optimierung wird immer die kürzeste Adressierung verwandt, wenn möglich. Dies läßt sich durch Ausschalten des Optimierers verhindern.

**(bd,An,Xn)**

Adressregister indirekt mit Index und Base-Displacement. Diese Adressierungsart arbeitet wie »a(An,Xn)«, nur kann das Base-Displacement »bd« hier bis zu 32 Bit groß sein oder fehlen, und das evtl. mit Scaling versehene Indexregister darf ebenfalls fehlen. Hier wird ggbf. zu »(An)« oder »a(An,Xn)« optimiert. »bd« darf absolut oder relativ sein und ist vorzeichenbehaftet. Beachten Sie, daß hierdurch auch z.B. »(Dn)« erlaubt ist, d.h. Datenregister indirekt, oder »()«, was Adresse 0 anspricht.

**(bd,PC,Xn)**

PC-relativ mit Index und Base-Displacement. Das oben Gesagte gilt auch hier, bezieht sich jedoch auf die PC-relativen Adressierungen. Hier wird ggbf. zu »r(PC)« oder »r(PC,Xn)« optimiert.

**([bd,An],Xn,od)**

Speicher indirekt post-indexed. Auch hier dürfen die einzelnen Bestandteile weggelassen werden, z.B. spricht »([])« die Adresse an, deren Zeiger in Adresse 0 steht. Das Outer-Displacement muß relativ oder importiert sein.

**([bd,An,Xn],od)**

Speicher indirekt pre-indexed. Hier wird die Indirektion erst nach der Indizierung durchgeführt.

**([bd,PC],Xn,od)**

PC-relativ Speicher indirekt post-indexed. Hier wird »bd« als PC-relativ angesehen und statt eines Adreßregisters der aktuelle PC addiert.

**([bd,PC,Xn],od)**

PC-relativ Speicher indirekt pre-indexed. Hier wird die Indirektion erst nach der Indizierung durchgeführt.

Bei den PC-relativen Modi ist zudem auch der String »ZPC« statt »PC« erlaubt. In dem Fall wird der PC nicht hinzuaddiert, die resultierende Adresse ist also absolut, und die Function-Code-Leitungen des Prozessors sind beim Zugriff auf Program-Space geschaltet. Dies werden Sie in der Regel nicht benötigen.

### 3.5.3 Spezialregister

Es gibt noch implizite Adressierungsarten, die ein Spezialregister des Prozessors, der PMMU oder FPU ansprechen:

Für die MPU:

**SP** Stackpointer

**SR** Status-Register

**CCR** Condition-Code-Register, die unteren 8 Bit des Statusregisters

**USP** User-Stackpointer (im Supervisor-Mode)

**VBR** Vector-Base-Register (ab MC68010)

**SFC** Source-Function-Code (ab MC68010)

**DFC** Destination-Function-Code (ab MC68010)

**CACR** Cache-Control-Register (ab MC68020)

**CAAR** Cache-Address-Register (nur MC68020 und MC68030)

**MSP** Master-Stackpointer (ab MC68020)

**ISP** Interrupt-Stackpointer (ab MC68020)

Für die PMMU (MC68851 oder ab MC68030; MC68040 hat andere PMMU-Kodierung):

**SRP** Supervisor-Root-Pointer

**CRP** CPU-Root-Pointer

**URP** User-Root-Pointer nur für MC68040, Synonym für CRP

**TC** Translation-Control-Register. Dieser Name kollidiert bei 1.3-Includefiles mit einem gleichen Namen in »exec/tasks.i«, ändern Sie daher fremde 1.3- Includefiles ab, indem Sie dort an »TC« einen Unterstrich hängen. Das dortige Symbol wird ohnehin nie verwendet und sorgt nur für einen Redefinitionsfehler. Bei den mitgelieferten 1.3-Includefiles ist diese Modifikation bereits durchgeführt.

**MMUSR** PMMU-Statusregister

**PSR** Synonym für MMUSR (wurde für MC68851 benutzt)

**TT0** Transparent-Translation-Register 0 (nur MC68030)

**TT1** Transparent-Translation-Register 1 (nur MC68030)

**DTT0** Data-Transparent-Translation-Register 0 (ab MC68040)

**DTT1** Data-Transparent-Translation-Register 1 (ab MC68040)

**ITT0** Instruction-Transparent-Translation-Register 0 (ab MC68040)

**ITT1** Instruction-Transparent-Translation-Register 1 (ab MC68040)

Für die FPU (MC68881, MC68882 oder ab MC68040):

**FPSR** FPU-Statusregister

**FPCR** FPU-Control-Register

**FPIAR** FPU-Instruction-Address-Register

**FPn** Fließkommaregister FP0 bis FP7



### 3.5.4 Adressierungs-Schablonen

Es folgt eine Zusammenfassung aller Adressierungsart-Formen, die erkannt werden:

| | | | |
|---|---|---|---|
| Dn | An | (An) | (An)+ |
| -(An) | d16(An) | (d8,An) | (d16,An) |
| (d32,An) | d8(An,Xn) | (An,Xn) | (d8,An,Xn) |
| (d16,An,Xn) | (d32,An,Xn) | (d8,Xn) | (d16,Xn) |
| (d32,Xn) | (Xn) | () | r16(PC) |
| (r8,PC) | (r16,PC) | (r32,PC) | r8(PC,Xn) |
| (r8,PC,Xn) | (r16,PC,Xn) | (r32,PC,Xn) | (r8,ZPC) |
| (r16,ZPC) | (r32,ZPC) | (r8,ZPC,Xn) | (r16,ZPC,Xn) |
| (r32,ZPC,Xn) | a | a.l | a.w |
| r | r.l | r.w | r8(Xn) |
| #a | #r | | |
| ([]) | ([],od) | ([],Xn) | ([],Xn,od) |
| ([An]) | ([An],od) | ([An],Xn) | ([An],Xn,od) |
| ([Xn]) | ([Xn],od) | ([An,Xn]) | ([An,Xn],od) |
| ([bd]) | ([bd],od) | ([bd],Xn) | ([bd],Xn,od) |
| ([bd,An]) | ([bd,An],od) | ([bd,An],Xn) | ([bd,An],Xn,od) |
| ([bd,Xn]) | ([bd,Xn],od) | ([bd,An,Xn]) | ([bd,An,Xn],od) |
| ([bd,PC]) | ([bd,PC],od) | ([bd,PC],Xn) | ([bd,PC],Xn,od) |
| ([bd,PC,Xn]) | ([bd,PC,Xn],od) | ([bd,ZPC]) | ([bd,ZPC],od) |
| ([bd,ZPC],Xn) | ([bd,ZPC],Xn,od) | ([bd,ZPC,Xn]) | ([bd,ZPC,Xn],od) |

## 3.6 Die Direktiven

Als Opcode erkennt der Assembler alle Instruktionen der Motorola-68000-Familie sowie Makros (siehe Unterkapitel 3.6.6 über Makros) und Direktiven. Zusätzlich existiert der ILLEGAL-Opcode, der die Kodierung $4AFC erzeugt. Direktiven und Instruktionen des Prozessors erkennt OMA ohne Case-Unterschied.

Bei den Prozessorinstruktionen prüft der Assembler, ob der Befehl auf eine gerade Adresse kommen wird, da sonst eine Adreß-Exception des Prozessors ausgelöst würde. Sollte dies der Fall sein, meldet der Assembler ein Warning und legt den Befehl auf eine gerade Adresse, d.h., der Programmzähler wird gerade gemacht. Achten Sie in dem Fall darauf, daß ein evtl. in der vorigen Zeile einzeln definiertes Symbol ungerade ist; ein Ansprung der Instruktion würde auf einer ungeraden Adresse landen!

OMA assembliert immer relokatiblen Code, d.h., es ist egal, wo das fertige Programm im Speicher zu liegen kommt. Daher ist der Programmzähler auch relativ, denn er gibt den Offset zum Programmstart an.

Direktiven sind Pseudobefehle des Assemblers, die genau wie Prozessorinstruktionen aussehen, jedoch ganz bestimmte Steuerungsaufgaben innerhalb des Assemblers haben. Im folgenden werden die einzelnen Direktiven nach Gruppen geordnet beschrieben. Für manche Direktiven gibt es Synonyme, die statt der üblichen Direktive auch verwandt werden können und Quelltextkompatibilität zu anderen Assemblern gewährleisten. Wenn Sie neue Quelltexte verfassen, sollten Sie aber die eigentlich vorgesehene Direktive verwenden. Ein Labelfeld ist für eine Direktive nur erlaubt, wenn es in der Formatschablone explizit angegeben ist. Ist *Label* nicht in eckige Klammern eingefaßt, dann muß ein Labelfeld vorhanden sein.

---

### 3.6.1 Steuerungsdirektiven

---

```
MC68000
MC68010
MC68020
MC68030
MC68040
MC68881
MC68882
```

Diese Direktiven wählen den passenden Prozessortyp oder schalten die FPU-Befehle ein. Es wird immer getestet, ob der gerade zu verarbeitende Befehl oder ein Register für den eingeschalteten Prozessor zulässig ist, z.B. gibt es einen Fehler, wenn Sie die MC68010 gewählt haben und die PACK-Instruktion assemblieren oder das CACR ansprechen möchten. Sie dürfen zu geringeren und höheren Stufen schalten, aber sobald einmal angewählt, können die FPU-Befehle immer verarbeitet werden; sie lassen sich danach nicht wieder abschalten.



Beachten Sie, daß Sie zum Assemblieren von Fließkomma-Ausdrücken eine FPU in Ihrem Rechner besitzen **müssen**, da OMA diese mit Hilfe der FPU auswertet.

Möchten Sie PMMU-Befehle benutzen, sollten Sie eine MPU ab MC68030 auswählen. Beachten Sie dabei aber, daß z.B. MC68030 und MC68040 völlig inkompatible PMMU-Befehle besitzen.

---

```
CPID   n
```

Da es mehrere Coprozessoren in einem Motorola-68000-System geben kann, können Sie mit dieser Direktive die Coprozessor-ID für FPU-Befehle verändern. »n« ist die neue Nummer, muß absolut sein, im Bereich von 0 bis 7 liegen und ist voreingestellt auf den Wert eins, d.h., es wird die normale FPU-ID angenommen.

In der Regel brauchen Sie dies nur, wenn Sie mehrere FPUs in einem System haben und diese parallel arbeiten lassen wollen. Bis jetzt ist ein solcher Fall noch auf keinem Amiga aufgetreten.

---

```
[Label]   SECTION   Name[,Typ][,Speichertyp]
```

*Bitte lesen Sie in diesem Zusammenhang auch das Kapitel 3.8 über den Beginn einer Sektion.* Der Assembler schreibt die erzeugten Daten in Sektionen, d.h. Sammeleinheiten für Daten. Daten in unterschiedlichen Sektionen sind funktionsmäßig getrennt und gelangen auch in getrennter Weise in das Objektfile. Es gibt drei Arten von Sektionen: Code (sollte immer Prozessorbefehle bekommen), Data (hier sollten Konstanten und Variablen mit Anfangswerten hineinkommen) und Bss (speichert Variablen ohne Initialwert und ist im Objektfile sehr kurz, da nur die Sektionslänge vermerkt ist). Es ist immer eine Sektion eingeschaltet; der Assembler beginnt mit einer namenlosen Code-Sektion und dem Programmzählerwert Null. Mit SECTION können Sie nun zwischen Sektionen hin- und herschalten. SECTION schaltet auf die Sektion mit dem Namen »Name« oder erzeugt sie, wenn sie nicht existiert. Es wird ab der letzten Programmzähler-Einstellung weiterassembliert. *Label* wird nach dem Umschalten dem neuen Programmzähler zugewiesen. Für *Name* gelten die üblichen Stringregeln.

*Typ* ist der Sektionstyp, erlaubt ist CODE, DATA oder BSS (kein Case-Unterschied). Wenn der Typ fehlt, wird eine Code-Sektion definiert. *Speichertyp* definiert, in welche Speicherbereiche des Amiga diese Sektion geladen werden kann. Erlaubt ist PUBLIC, CHIP oder FAST. Alternativ kann für »CODE,CHIP« bzw. »CODE,FAST« auch »CODE_C« bzw. »CODE_F« benutzt werden, analog auch DATA_C, DATA_F, BSS_C und BSS_F.

Möglich sind bis zu 255 Sektionen pro Assemblerlauf; falls die gewünschte Sektion bereits existiert und bei SECTION der Sektionstyp und evtl. der Speichertyp angegeben ist, müssen diese beiden mit der ersten Definition dieser Sektion übereinstimmen, sonst erfolgt eine Fehlermeldung.

---

`[Label]  CODE`

Diese Direktive arbeitet wie SECTION, schaltet aber auf die namenlose Code-Sektion mit dem Speichertyp PUBLIC. *Label* wird nach dem Umschalten zugewiesen.

---

`[Label]  DATA`

Diese Direktive arbeitet wie SECTION, schaltet aber auf die namenlose Data-Sektion mit dem Speichertyp PUBLIC. *Label* wird nach dem Umschalten zugewiesen.

---

`[Label]  BSS`

Diese Direktive arbeitet wie SECTION, schaltet aber auf die namenlose Bss-Sektion mit dem Speichertyp PUBLIC. *Label* wird nach dem Umschalten zugewiesen.

---

`[Label]  RORG  Absolutausdruck`

Der Programmzähler wird auf *Absolutausdruck* gesetzt. *Label* wird nach dem Umschalten zugewiesen.

*Absolutausdruck* darf nicht negativ oder kleiner als der Programmzähler sein. Diese Inkompatibilität zum Metacomco-Assembler ist aber kein Hindernis, da beim Rücksetzen des Programmzählers die alten, schon erzeugten Daten und Instruktionen durch die neuen überschrieben würden. Ein Rücksetzen des Programmzählers ist daher ohnehin sinnlos. Sollte der neue Programmzähler hinter dem alten liegen, wird der Zwischenraum durch Nullbytes aufgefüllt.

Synonym: RORIGIN.

---

`ORG  Absolutausdruck`

Dieser Befehl wird nicht unterstützt und bewirkt eine Fehlermeldung. Für das Erzeugen absoluter EPROM-Programme sei auf das Hilfsprogramm *Absolute* verwiesen. Nachteil: Die Adressierungsart »Absolut kurz« wird nicht in allen Fällen automatisch als Optimierung verwendet. Dies sollte aber nicht ins Gewicht fallen, da beim Amiga diese Adreßbereiche ohnehin nicht für EPROMs geeignet sind.

Synonym: ORIGIN, ABSOLUTE.

---



`[Label]  CNOP  Offsetausdruck,Basisausdruck`

Der Programmzähler kann mit CNOP auf eine beliebige Grenze gelegt werden. Er wird auf die nächste Grenze der Basisnummer plus dem Offset gesetzt. *Offsetausdruck* und *Basisausdruck* müssen absolut und ohne Vorwärtsreferenz sein. *Label* wird nach dem Umschalten zugewiesen. Beispiele:

```
CNOP   0,4  legt den Programmzähler auf die nächste Langwortgrenze
CNOP   0,2  legt den Programmzähler auf die nächste Wortgrenze
```

Der Programmzähler sei 6. Bei "CNOP 3,4" ist 8 die nächste Langwortgrenze und 8+3=11 die Adresse mit dem Offset. Da aber 11-4=7 noch hinter 6 liegt, wird der neue Programmzähler 7 sein!

Der freie Raum zwischen neuem und altem Programmzähler wird mit Nullbytes aufgefüllt. In einem OFFSET-Abschnitt (siehe unten) wird der Offsetzähler statt des Programmzählers geändert.

---

`[Label]  EVEN`

Der Programmzähler wird auf die nächste gerade Adresse gesetzt. *Label* wird nach dem Umschalten zugewiesen. Der freie Raum zwischen neuem und altem Programmzähler wird mit Nullbytes aufgefüllt.

In einem OFFSET-Abschnitt (siehe unten) wird der Offsetzähler statt des Programmzählers geändert.

---

`BASEREG  Adreßregister`

Mit BASEREG kann der Basereg-Modus ein- oder ausgeschaltet werden. Wenn der String *Adreßregister* der String »OFF« (kein Case-Unterschied) ist, wird der Modus ausgeschaltet.

Der Basereg-Modus ermöglicht es, in einem Programm, das nur aus einem einzigen Quelltext besteht, die verkürzte »a(An)«-Adressierungsart aus »Absolut Lang« zu erzeugen, sofern die Adresse in einer Bss-Sektion liegt. Dazu sollte nur eine Bss-Sektion definiert und ein Adreßregister als Basiszeiger reserviert werden. Der Basiszeiger **muß** die Adresse »Bss-Sektion-Start + 32768« enthalten!

Die Werte der Bss-Sektion müssen vor ihrer Benutzung bekannt sein, sonst kann der Assembler die Adressierungsart nicht konvertieren (oder es muß der Mehrpass-Modus eingeschaltet werden). Ein Beispiel:

```
   CODE                  ; schalte auf namenlose Code-Sektion
   lea   Basis+32768,a4  ; initialisiere Basisregister richtig und
                         ; aktiviere die Code-Sektion als erste
   BASEREG  a4           ; schalte Basereg-Modus ein

   BSS                   ; schalte auf namenlose Bss-Sektion
Basis                    ; erste Adresse der Bss-Sektion
wert1 ds.l  1            ; wert1 = 0 relativ
wert2 ds.l  1            ; wert2 = 4 relativ

   CODE                  ; schalte auf Code-Sektion zurück
   move  d0,wert2        ; wird als "MOVE D0,-$7FFC(A4)" direkt
                         ; hinter obiges "lea" assembliert
   clr   wert1           ; wird als "CLR -$8000(A4)" assembliert

   BASEREG  off          ; schaltet den Basereg-Modus wieder aus
```

Die Code-Sektion muß vor der Bss-Sektion angesprochen werden, da sonst die Bss-Sektion als erster Hunk im Objektfile steht und die Ausführung des Programmes in ihr beginnen würde (siehe Kapitel 3.8)! Mit dieser Adressierung kann eine Bss-Sektion von maximal 64 KByte Größe adressiert werden. Variablen über dieser Grenze werden immer »Absolut Lang« assembliert.

Der Linker erfährt von diesem Modus nichts; eine Veränderung des Basisregisters durch z.B. »move.l d0,a4« während des Basereg-Modus führt zu einem Warning.

---

`[Label]  END`

Beendet den laufenden Assembler-Pass und ignoriert alle im Quellfile folgenden Zeilen. *Label* wird vor dem Beenden zugewiesen. END darf nicht in einem Includefile vorkommen (siehe INCLUDE-Direktive), und jeder Quelltext sollte mit END enden, da sonst ein Warning generiert wird.

Synonym: ENDSRC.

---

`INCLUDE  Filename`

Der Assembler unterbricht die Assemblierung des aktuellen Quellfiles und schiebt das Includefile Filename in den Assemblerlauf ein. Wenn dieses neue File zu Ende ist, wird im alten Quellfile fortgefahren. Mit INCLUDE werden z.B. Konstanten für den Amiga aus dem Include-Directory gelesen. Es wird zuerst im aktuellen Directory nach dem File gesucht, dann in den beim Assembleraufruf mit der »-i«-Option angegebenen Directories, dann in den INCDIR-Directories (siehe folgende Direktive) sowie im evtl.



vorhandenen Default-Includedirectory AINC: (siehe auch »-i«-Startoption). Für Filename gelten die üblichen Stringregeln. Sie dürfen Includefiles beliebig tief verschachteln.

Bitte beachten Sie, daß Sie durch Vorassemblieren von Includefiles Zeit sparen können. Siehe Kapitel 3.13.

Synonym: GET.

---

`INCDIR  Name[,Name]...`

Diese Direktive fügt der Liste der Directories zum Suchen von Includefiles neue Namen hinzu. Für *Name* gelten die üblichen Stringregeln. Bitte beachten Sie, daß das Default-Includedirectory AINC: hinter den mit INCDIR definierten Directories in der Liste steht, falls es existiert. Das heißt, daß es dann nach den INCDIR-Namen nach Includefiles durchsucht wird.

---

`[Label]  INCBIN  Filename[,Byteanzahl]`

Der Assembler weist *Label* als relatives Symbol dem Programmzähler zu und liest aus dem File Filename maximal *Byteanzahl* Binärbytes. Wenn *Byteanzahl* fehlt, werden alle Bytes des Files eingelesen. Es erfolgt keine Meldung, wenn weniger als *Byteanzahl* Bytes gelesen wurden. Filename gehorcht den üblichen Stringregeln und muß vollständig vorhanden sein, da die Include-Directory-Liste hier nicht benutzt wird.

Synonym: IBYTES.

---

`[Label]  INCPRE  Filename`

INCPRE inkludiert ein vorassembliertes File genau wie die Startoption »-p«, kann allerdings mehrfach verwendet werden. Damit können mehrere unabhängige, vorassemblierte Files in den Assemblerlauf einbezogen werden bzw. man kann sich die Startoption sparen. Die Symbole in den vorassemblierten Files dürfen nicht andere redefinieren, und der Assembler ist ebenso empfindlich gegenüber falschen Files wie bei der Startoption. Für Filename gelten die üblichen Stringregeln. Ein Label ist erlaubt. Siehe auch das Kapitel über vorassemblierte Includefiles.

---

`OUTPUT  Name`

Mittels OUTPUT können Sie die Namensgebung für das Objektfile beeinflussen. *Name* darf mit einem Punkt beginnen, in dem Fall zählt *Name* als Suffix für den Objektfilenamen. Wenn schon mittels »-o«-Option ein Name angegeben wurde, bekommt das Objektfile diesen Namen. Sonst gelten die folgenden Regeln:

- Ist *Name* kein Suffix, bekommt das Objektfile den Namen Name.
- Ist *Name* ein Suffix, bekommt das Objektfile den Namen, der sich aus dem Quellfilenamen ohne evtl. ≫.asm≪-Suffix, aber mit *Name* als Suffix zusammensetzt.

Wurde weder die ≫-o≪-Option noch OUTPUT benutzt, dann gilt:

- Ist das Programm direkt ausführbar, erhält es den Namen, der sich aus dem Quellfilenamen ohne evtl. ≫.asm≪-Suffix zusammensetzt.
- Ist das Objektfile nur linkbar, bekommt es den Namen, der sich aus dem Quellfilenamen ohne evtl. ≫.asm≪-Suffix, aber mit ≫.obj≪ als Suffix zusammensetzt.

---

```
OPT   Option[,Option]...
```

OPT ermöglicht die Auswahl mehrerer Optionen. Jede Option beginnt mit einem Buchstaben, gefolgt von ≫+≪ oder ≫-≪. Es gibt folgende Optionen:

**C+** Es wird der Case-Unterschied gemacht. Mit C- wird nicht unterschieden. Diese Option darf mehrfach vorkommen und hat Priorität vor der ≫-c c≪-Startoption.

**D+** Relative Symbole werden ins Objektfile in einen Symbolhunk geschrieben. Diese Option entspricht der ≫-c s≪-Startoption, ist aber wirkungslos, falls diese verwandt wurde. D- verhindert die Symbolhunkproduktion. Die letzte Einstellung ist gültig.

**L+** bestimmt, daß ein linkbares Objektfile erzeugt wird. L- bestimmt, daß ein ausführbares Programm erzeugt wird. Bei XREF/XDEF wird dann ein Fehler erzeugt. *Bitte* beachten Sie, daß diese Option *auf der allerersten Zeile* stehen muß und daß OMA selber zwischen ausführbar und linkbar entscheidet (anhand von Vorkommen/Fehlen von XREF/XDEF), falls die Option nicht benutzt wird.

**M+** entspricht etwa MLIST, M- entspricht etwa NOMLIST. Der Unterschied besteht darin, daß die ≫-c m≪-Startoption Vorrang vor M+ und M- hat, aber MLIST und NOMLIST Vorrang vor der Startoption.

**O+** schaltet den Optimierer ein, O- schaltet ihn aus. Es ist nicht möglich, einzelne Optimierungsstufen ein- bzw. auszuschalten. Die Option hat Priorität gegenüber der ≫-c o≪-Startoption. Siehe auch OPTIMIZE/NOOPTIM-Direktiven.

**P+** bewirkt, daß in jeder Zeile mit einer Relokation ein Fehler erzeugt wird, was Sie zum Erstellen von positionsunabhängigen, d.h. vollständig PC-relativen Programmen benutzen können. P- schaltet diesen Modus wieder aus.

**S+** listet nach dem Listing die Symboltabelle ins Listingfile, S- verhindert dies. Die ≫-c t≪-Startoption hat Vorrang vor dieser Option. Die jeweils letzte Einstellung ist gültig.

**U+** Die speziellen lokalen Label werden ab sofort mit einem Unterstrich ≫_≪ statt mit einem Punkt ≫.≪ eingeleitet. Mit U- werden die speziellen lokalen Label ausgeschaltet, d.h., nur noch Label der Form ≫1$≪, ≫45$≪ usw. gelten als lokal.



**W+** schaltet das Ausgeben von Warnings an, W- schaltet es aus. Die ≫-c w≪-Startoption hat Vorrang vor dieser Option.

**X+** Es werden nur exportierte, relative Symbole in einen Symbolhunk geschrieben. Die ≫D+≪-Option und die ≫-c s≪-Startoption haben Vorrang, bei ihnen werden alle relativen Symbole geschrieben, auch wenn X+ eingeschaltet wurde. X- schaltet den Modus aus. Die jeweils letzte Einstellung ist gültig.

---

```
[Label]   REPT   Anzahl
          (Befehle)
          ENDR
```

Mit diesen Direktiven fassen Sie mehrere Befehle zusammen und wiederholen diesen Block *Anzahl* mal. *Anzahl* muß absolut und ohne Vorwärtsreferenz sowie größer als Null sein. Die Direktiven REPT und ENDR sind nicht in Makros erlaubt, ebenso darf bei den Befehlen keine Makroaufnahme vorkommen, aber eine Makroexpansion ist erlaubt. Beispiel zum Erzeugen von zehn NOPs:

```
REPT   10
nop
ENDR
```

---

```
OPTIMIZE
```

Der Optimierer wird eingeschaltet. Diese Direktive kann die Aufrufoption ≫-c o≪ unwirksam machen. Siehe Kapitel 3.9.

Synonym: OPT O+.

---

```
NOOPTIM
```

Der Optimierer wird wieder ausgeschaltet. Siehe Kapitel 3.9.

Synonym: OPT O-.

---

```
NOOBJ
```

Diese Direktive hat dieselbe Wirkung wie die Option ≫-c n≪. Die Produktion des Objektfiles wird unterdrückt.

---

```
FAIL
```

Meldet einen benutzererzeugten Fehler in der aktuellen Zeile. Diese Direktive wird im allgemeinen benutzt, um auf Fehler in Makros hinzuweisen. Siehe Unterkapitel 3.6.6.

---

```
MASK2
FORMAT
NOFORMAT
COMMENT
SUBTTL
```

Diese Direktiven werden akzeptiert, haben aber keine Wirkung und werden nur aus Kompatibilitätsgründen zu anderen Assemblern verstanden.

---

### 3.6.2 Zuweisungsdirektiven

---

```
Label   EQU   Ausdruck
```

Es wird ein Symbol namens *Label* mit dem Wert *Ausdruck* definiert. Der Ausdruck darf keine Vorwärtsreferenzen haben (d.h. Symbole benutzen, die in einer späteren Zeile definiert werden) und kann relativ, absolut oder importiert sein. Das neue Symbol darf noch nicht existieren.

Synonym: EQUATE und das Gleichheitszeichen »=«.

---

```
Label   SET   Ausdruck
```

Diese Direktive arbeitet wie EQU, nur kann das Symbol mit SET redefiniert werden, d.h. einen neuen Wert bekommen. Ein SET-Symbol darf nicht vor seiner Definition verwendet werden oder schon durch ein EQU definiert sein. Siehe auch Unterkapitel 3.6.6 über Makros.

Folgende Assembler-Symbole sind reserviert und können nicht redefiniert werden (kein Case-Unterschied):

```
D0  D1  D2  D3  D4  D5  D6  D7  A0  A1  A2  A3  A4  A5  A6  A7
FP0 FP1 FP2 FP3 FP4 FP5 FP6 FP7 PC  SP  SR  CCR USP VBR SFC DFC
CACR CAAR MSP  ISP  SRP  CRP  URP  TC   TT0  TT1  DTT0 DTT1 ITT0 ITT1 PSR
FPCR FPSR NARG __RS MMUSR    FPIAR    MACRO    MEXIT    ENDM ENDMAC
```



Synonym: SETVAL.

---

```
Label   FEQU[.s]   FließkommaAusdruck
```

Es wird ein Fließkommasymbol namens *Label* mit dem Wert *Ausdruck* definiert. Als Size-Specifier ist S, D, X, P oder keiner zugelassen, dieser hat jedoch keine Auswirkung, da der Wertebereich immer Extended-Precision ist. Der Ausdruck darf keine Vorwärtsreferenzen haben und ist ein Fließkomma-Ausdruck, darf also andere Fließkommasymbole enthalten. Das neue Symbol darf noch nicht existieren. Fließkommasymbole können z.B. in der folgenden Form verwendet werden:

```
   mc68882                          ; oder mc68881; schalte erst FPU ein
Pi FEQU    3.14159265358979323846   ; definiere Pi
   fmove.x #Pi,fp2                  ; normaler Gebrauch
   fmul.w  #3*Pi,fp2                ; multipliziere mit dreifachem Pi
```

---

```
Label   FSET[.s]   FließkommaAusdruck
```

Diese Direktive arbeitet wie FEQU, nur kann das Symbol mit FSET redefiniert werden. Ein FSET-Symbol darf nicht vor seiner Definition verwendet werden oder schon durch ein FEQU definiert sein. Für die Liste der nicht redefinierbaren Symbolnamen siehe SET.

---

```
Label   EQUR   Register
```

Es wird ein Registersymbol namens *Label* für das Register *Register* definiert. *Register* darf D0 bis D7 oder A0 bis A7 sein. Ein Registersymbol kann im Quelltext ohne Case-Unterschied überall dort verwandt werden, wo ein Daten- oder Adreßregister verlangt wird. Es darf aber nicht vor seiner Definition angesprochen werden.

---

```
Label   SETR   Register
```

Diese Direktive arbeitet wie EQUR, nur kann das Symbol mit SETR redefiniert werden, d.h. ein neues Register repräsentieren. Ein SETR-Symbol darf nicht vor seiner Definition verwendet werden oder schon durch ein EQUR definiert sein. Für die Liste der nicht redefinierbaren Symbolnamen siehe SET.

---

```
Label   REG   Registermaske
```

Der Assembler definiert das Symbol namens *Label* als Registermaske für den Gebrauch mit der Prozessorinstruktion MOVEM. *Registermaske* hat die Syntax

`Dn[-Dn][/Dn][/An][-An]`

Beispiel **d0-d3/d7/a0-a4**: Die Maske besteht aus den Registern D0, D1, D2, D3, D7, A0, A1, A2 und A4. Dies ließe sich auch als **d0-d3/d7-a4** schreiben.

Weitere Beispiele: `a0/a2 a0/d0 a0/a2-a3/d5-d7`

Als Registermaske ist auch ein Symbol erlaubt, das mit einer vorherigen REG-Direktive deklariert wurde.

Synonym: EQURL.

---

```
Label   SETREG   Registermaske
```

Diese Direktive arbeitet wie REG, nur kann das Symbol mit SETREG redefiniert werden. Ein SETREG-Symbol darf nicht vor seiner Definition verwendet werden oder schon durch ein REG definiert sein. Für die Liste der nicht redefinierbaren Symbolnamen siehe SET.

Synonym: SETRL.

---

```
IDNT   String
```

Ein Objektfile enthält einen Unitnamen, der bis zur Verwendung dieser Direktive leer ist. Der neue Name ist *String* und gehorcht den üblichen Stringregeln.

Synonym: IDENTIFY.

---

```
XDEF   Label[,Label]...
```

XDEF macht relative oder absolute Symbole für andere Objektfiles sichtbar und vermerkt, daß das Objektfile linkbar sein wird und nicht direkt ausführbar. Andere Objektfiles können sich durch XREF auf durch XDEF extern sichtbar gemachte Symbole beziehen. Der Linker hat die Aufgabe, diese Referenzen aufzulösen. Für eine genauere Beschreibung zum Benutzen der beiden Direktiven siehe XREF unten.

Falls »OPT L-« eingeschaltet ist, wird bei dieser Direktive ein Fehler erzeugt.

Synonym: EXPORT (besonders empfohlen), ENTRY.

---



```
XREF   Label[,Label]...
```

XREF definiert importierte Symbole, die relativ oder absolut sein können, und vermerkt, daß das Objektfile linkbar sein wird und nicht direkt ausführbar. Wie oben erwähnt, können damit Referenzen auf andere Objektfiles vereinbart werden.

Falls »OPT L-« eingeschaltet ist, wird bei dieser Direktive ein Fehler erzeugt.

Synonym: IMPORT (besonders empfohlen), EXTRN, EXTERN.

XREF und XDEF werden verwendet, um große Programme zu modularisieren. Sie deklarieren z.B. in einem Modul A (d.h. in einem Textfile, vielleicht mit dem Namen text1.asm) eine Routine namens »KopiereString«. Wenn Sie in diesem Modul A den Befehl »XDEF KopiereString« geben, dann macht der Assembler diesen Namen für andere Module sichtbar. Sie benötigen jetzt z.B. in einem Modul B (d.h. in einem Textfile namens z.B. bearbeite.asm) genau diese Routine »KopiereString« und können nun diese Routine aus Modul A verwenden, indem Sie »XREF KopiereString« befehlen. Der Assembler assembliert jetzt zwar nicht erneut eine Routine »KopiereString«, aber er vermerkt, daß diese Routine aus Modul A in Modul B benutzt wurde. Die beiden Objektfiles text1.obj und bearbeite.obj, die der Assembler erzeugt hat, müssen mit dem Linker zu einem Programm zusammengelinkt werden, wobei der Linker die Referenz in B auf »KopiereString« aus A auflöst. Beachten Sie, daß der Assembler bei Fehlen von XREF meldet, er kenne kein Symbol »KopiereString«, während er das Fehlen von XDEF nicht bemerken kann. Fehlt aber XDEF, dann wird der Linker melden, daß er die Referenz nicht lösen kann, da er »KopiereString« nicht sieht (denn es wurde nicht sichtbar gemacht).

EXPORT bzw. IMPORT sind Synonyme für XDEF bzw. XREF und sollten statt diesen verwendet werden, da sie aussagekräftiger sind und den gleichnamigen Modula-2-Befehlen entsprechen.

---

```
[Label]   DC[.size]   Ausdruck[,Ausdruck]...
[Label]   DC.S        FließkommaAusdruck[,FließkommaAusdruck]...
[Label]   DC.D        FließkommaAusdruck[,FließkommaAusdruck]...
[Label]   DC.X        FließkommaAusdruck[,FließkommaAusdruck]...
[Label]   DC.P        FließkommaAusdruck[,FließkommaAusdruck]...
```

Der Assembler weist *Label* als relatives Symbol dem Programmzähler zu und speichert Ausdrücke. Abhängig von *size* werden Bytes, Worte oder Langworte gespeichert. Mit den übrigen Größenangaben können Sie auch Fließkommakonstanten unterschiedlicher Genauigkeit in den Code einfügen.

Falls es die Bytegröße ist, ist als Ausdruck noch eine Folge von Zeichen erlaubt, die in einfache oder doppelte Hochkommata eingefaßt ist und als String gespeichert wird. Soll dann das Hochkomma selber als Zeichen erscheinen, muß es zweimal angegeben werden. Beispiele:

```
dc.b  'Dies ist ein String',0
dc.b  "Dies ist ein String",0
dc.b  'Dies ist''n String',0
dc.b  "Dies ist'n String",0
dc.b  'Dies ist"n String',0      (falsches Deutsch)
dc.b  "Dies ist""n String",0     (falsches Deutsch)
```

Bei Bytegröße darf der Ausdruck nicht relativ sein; ist es keine Bytegröße, dann wird der Programmzähler noch vor der Labelzuweisung gerade gemacht, um Adreß-Exceptions des Prozessors MC68000 oder MC68010 zu verhindern.

Synonyme: DB=DC.B, DW=DC.W, DL=DC.L, DD=DC.D, DX=DC.X, DP=DC.P.

---

```
[Label]   DCB[.size]   Absolutausdruck,Ausdruck
[Label]   DCB.S        Absolutausdruck,FließkommaAusdruck
[Label]   DCB.D        Absolutausdruck,FließkommaAusdruck
[Label]   DCB.X        Absolutausdruck,FließkommaAusdruck
[Label]   DCB.P        Absolutausdruck,FließkommaAusdruck
```

Ist es keine Bytegröße, so wird der Programmzähler gerade gemacht. Danach wird *Label* als relatives Symbol dem Programmzähler zugewiesen und ein Block von *Absolutausdruck* mal dem *Ausdruck* erzeugt. Abhängig von *size* werden Bytes, Worte oder Langworte gespeichert. Bei Bytegröße darf der Ausdruck nicht relativ sein. Mit den übrigen Größenangaben können Sie auch Blöcke von Fließkommakonstanten unterschiedlicher Genauigkeit in den Code einfügen. Beispiel:

```
dcb.w 4,2      entspricht    dc.w 2,2,2,2
dcb.s 4,3.14   entspricht    dc.s 3.14,3.14,3.14,3.14
```

Synonym: BLK.

---

```
[Label]   DS[.size]   Absolutausdruck
[Label]   DS.S        Absolutausdruck
[Label]   DS.D        Absolutausdruck
[Label]   DS.X        Absolutausdruck
[Label]   DS.P        Absolutausdruck
```

Diese Direktive wird zum Reservieren von Speicher oder zum Zuweisen von Konstanten im Offsetabschnitt gebraucht.

Falls KEIN Offset-Abschnitt eingeschaltet ist:

> Ist es keine Bytegröße, wird der Programmzähler erst einmal gerade gemacht. *Label* wird als relatives Symbol dem Programmzähler zugewiesen. Danach stellt der Assembler den Programmzähler um *Absolutausdruck* Bytes, Worte, Langworte oder um die Größe der Fließkommazahl weiter. Der Zwischenraum wird mit Nullbytes gefüllt.



Falls ein Offset-Abschnitt eingeschaltet ist:

> Ist es keine Bytegröße, wird der Programmzähler erst einmal gerade gemacht. *Label* wird als **absolutes (!)** Symbol dem Offsetzähler zugewiesen und der Offsetzähler um *Absolutausdruck* Bytes, Worte, Langworte oder um die Größe der Fließkommazahl weitergestellt. Siehe OFFSET-Direktive.

Synonym: RESERVE.

---

```
          RSRESET
          RSSET       Absolutausdruck
[Label]   RS[.size]   Absolutausdruck
[Label]   RS.S        Absolutausdruck
[Label]   RS.D        Absolutausdruck
[Label]   RS.X        Absolutausdruck
[Label]   RS.P        Absolutausdruck
```

Mit diesen drei Direktiven ist es möglich, fortlaufend absolute Werte wie mit EQU zuzuweisen. Es wird ein interner Zähler mitgeführt:

```
  rsreset        ; setze Zähler auf Null
a rs.l  1        ; weise a=0 zu und stelle Zähler um 4 (Langwort) weiter
b rs.w  1        ; weise b=4 zu und stelle Zähler um 2 (ein Wort) weiter
  rsset 6        ; setze Zähler auf sechs
c rs    0        ; weise c=4 zu und stelle Zähler um 0 Worte weiter
                 ; (bleibt also unverändert)
```

Bitte beachten Sie, daß der Zähler vor der Zuweisung gerade gemacht wird, wenn ist nicht Bytegröße ist.

Die RS-Direktiven sind komfortabler als die Verwendung von OFFSET (siehe folgend), und das reservierte Symbol __RS (es sind zwei Unterstriche) enthält immer den aktuellen Wert des RS-Zählers. Betrachten Sie auch die Includefiles, dort wird von RS ausgiebig Gebrauch gemacht.

---

```
OFFSET   Absolutausdruck
```

Der Assembler beginnt einen Offsetabschnitt zum Zuweisen von Offsets ähnlich wie mit den RS-Direktiven. Der Offsetzähler wird auf *Absolutausdruck* gesetzt. Beispiel:

```
   OFFSET   0      ; beginne Offsetabschnitt mit Zähler=0
a  ds.l  1         ; weise a=0 zu und stelle Zähler um 4 (Langwort) weiter
b  ds.w  1         ; weise b=4 zu und stelle Zähler um 2 (ein Wort) weiter
   RORG  *         ; beende den Offsetabschnitt
```

Der Offsetabschnitt muß mit RORG, ORG, OFFSET, SECTION, CODE, DATA, BSS oder END beendet werden. In einem Offsetabschnitt sind keine codeproduzierenden Direktiven erlaubt, damit die DS-Direktive eindeutig entweder einem Offsetabschnitt oder einer Sektion zugeordnet werden kann.

Diese Direktive existiert aus Kompatibilitätsgründen, bitte benutzen Sie die RS-Direktiven.

---

### 3.6.3 Objektorientierte Direktiven

---

```
[Label]   METHOD   Objekt,MethodenID
```

Mit dieser Direktive wird eine Methode eines Objekts aufgerufen. *Objekt* darf dabei jede Adressierungsart sein (z.B. A5) und ist der Zeiger auf das Objekt. *MethodenID* ist eine Methoden-ID, deklariert mit Hilfe der mitgelieferten Makros im Includefile ooma.i. METHOD darf im Gegensatz zu SMETHOD an beliebigen Stellen im Programm auftreten. Beispiel:

```
        ; A5 zeige auf Objekt "Benji" der Klasse "Hund"
        METHOD  a5,Hund_Belle
        METHOD  tom,Katze_AufDenBaum
        ...
tom     ds.l    1               ; Zeiger auf Objekt der Klasse "Katze"
```

*Nach der METHOD-Direktive ist das Register A6 verändert!* Die objektorientierte Programmierung mit OMA wird ausführlich im Kapitel 7 behandelt.

---

```
[Label]   SMETHOD   Objekt,AktuelleKlasse,MethodenID
```

Diese Direktive ruft eine Methode eines Objektes auf, allerdings wird die Methodenimplementierung der Oberklasse angesprungen. SMETHOD wird nur in Methodenimplementierungen von Klassen verwendet und erlaubt es, die Funktionalität der Oberklasse zu erreichen. Ein einfacher METHOD-Aufruf würde nämlich nur die Methode derselben Klassenebene aufrufen, und dies führte zu einer Rekursion, wenn die aufrufende Methode den gleichen Namen hat. Beispiel:



```
        ; Eingabe: A5: Self.
METHODBODY Rechteck_Initialize
        PROC
        SMETHOD a5,ClassRechteck,Rechteck_Initialize  ; erst Oberklasse
        move    #1,rechteck_Höhe(a5)    ; jetzt Höhe auf 1
        move    #1,rechteck_Breite(a5)  ; und Breite auch
        ENDPROC
```

*Nach der SMETHOD-Direktive ist das Register A6 verändert!* Die objektorientierte Programmierung mit OMA wird ausführlich im Kapitel 7 behandelt. Dort wird auch gezeigt, wie mit Hilfe eines Makros namens SUPER die Angabe der aktuellen Klasse automatisiert werden kann.

---

### 3.6.4 Listingdirektiven

Für die folgenden Direktiven sollten Sie auch das Kapitel 3.12 über das Listingfile lesen.

---

```
PAGE
```

Gibt einen Seitenvorschub ins Listingfile und schaltet den Seitenvorschub wieder ein. Die Direktive erscheint selber nicht im Listing.

Synonym: PAGEUP.

---

```
NOPAGE
```

Schaltet den Seitenvorschub aus und erscheint selber nicht im Listing.

---

```
SPC   Nummer
```

Gibt *Nummer* Leerzeilen ins Listing und erscheint selber nicht im Listing. *Nummer* muß eine Dezimalzahl im Bereich 1 bis 250 sein.

Sofern der Seitenvorschub eingeschaltet ist, wird er bei den mit SPC erzeugten Zeilen weiterhin ausgelöst.

---

```
LLEN   Nummer
```

Die Zeilenlänge des Listings wird auf *Nummer* Zeichen eingestellt. Damit wird jede Listingzeile spätestens nach dem *Nummer*-ten Zeichen beendet. Die Direktive darf nur einmal pro Assemblerlauf vorkommen und erscheint nicht im Listing. *Nummer* muß eine Dezimalzahl im Bereich 60 bis 300 sein. Der Wert 132 ist voreingestellt.

---

`PLEN  Nummer`

Die Seitenlänge des Listings wird auf *Nummer* Zeilen eingestellt. Damit wird alle *Nummer* Zeilen ein Seitenvorschub ins Listing gegeben, wenn dieser nicht durch NOPAGE ausgeschaltet ist. Die Direktive darf nur einmal pro Assemblerlauf vorkommen und erscheint nicht im Listing. *Nummer* muß eine Dezimalzahl im Bereich 24 bis 100 sein. Der Wert 60 ist voreingestellt.

---

`TTL  Titel`

Setzt den Titel im Seitenvorschub des Listings auf *Titel* und sollte nur einmal pro Assemblerlauf vorkommen. Dieser Titel ist vorerst immer leer. Für *Titel* gelten die üblichen Stringregeln.

Synonym: TITLE.

---

`LIST`

Schaltet das Listen von Zeilen ein und erscheint selber nicht im Listing.

---

`NOLIST`

Schaltet das Listen von Zeilen aus und erscheint selber nicht im Listing. Sie können also verhindern, daß bestimmte Abschnitte ins Listing gelangen.

Synonym: NOL

---

`CLIST`

Schaltet das Listen von Zeilen innerhalb von Abschnitten der bedingten Assemblierung mit Falsch-Wert ein. Dieser Wert ist voreingestellt. Siehe auch IF-Direktiven.

---



`NOCLIST`

Schaltet das Listen von Zeilen innerhalb von Abschnitten der bedingten Assemblierung mit Falsch-Wert aus. Siehe auch IF-Direktiven.

---

`MLIST`

Schaltet das Listen von Zeilen während der Makroexpansion ein. Dieser Wert ist voreingestellt. Siehe auch 3.6.6.

---

`NOMLIST`

Schaltet das Listen von Zeilen während der Makroexpansion aus. Siehe auch 3.6.6.

---

`LISTCHAR  Ausdruck[,Ausdruck]...`

Schreibt ein oder mehrere Steuerzeichen mit dem ASCII-Wert *Ausdruck* in das Listingfile, sofern dieses nicht auf das Terminal geht. *Ausdruck* muß Bytegröße haben und absolut und positiv sein.

---

### 3.6.5 Bedingte Assemblierung

Die bedingte Assemblierung ermöglicht es, abhängig von einem Wahrheitswert einer IF-Direktive Quelltext-Teile von der Assemblierung auszuschließen oder sie einzubeziehen. IF-Direktiven können beliebig tief verschachtelt werden. Zwei Beispiele:

a)

```
ifeq  0        ; wahr, da 0=0
   nop         ; wird assembliert
   ifeq  1     ; falsch, da 1 ungleich 0
      rts      ; wird daher nicht assembliert
   endc
endc
```

b)

```
ifne  0          ; falsch, da 0 nicht ungleich 0
   nop           ; wird daher nicht assembliert
   ifne  1       ; wahr, da 1 ungleich 0
      rts        ; würde assembliert, aber die äußere Schleife..
   endc          ; ..ist falsch, also ist der Wahrheitswert der..
endc             ; ..inneren Schleife egal
```

Für die bedingte Assemblierung existieren die folgenden Direktiven:

---

`IFC  String1,String2`

Liefert Wahr, wenn *String1* und *String2* identisch sind. Für die Strings gelten der Case-Unterschied und die üblichen Stringregeln.

---

`IFNC  String1,String2`

Liefert Wahr, wenn die beiden Strings *nicht* identisch sind.

---

`IFD  Symbol`

Liefert Wahr, wenn das *Symbol* vor dieser Zeile definiert wurde.

---

`IFND  Symbol`

Liefert Wahr, wenn das *Symbol nicht* vor der Zeile definiert wurde.

---

`IFEQ  Absolutausdruck`

Liefert Wahr, wenn *Absolutausdruck* gleich Null ist.

---

`IFNE  Absolutausdruck`

Liefert Wahr, wenn *Absolutausdruck ungleich* Null ist.

---



`IFGT  Absolutausdruck`

Liefert Wahr, wenn *Absolutausdruck* größer als Null ist.

---

`IFLT  Absolutausdruck`

Liefert Wahr, wenn *Absolutausdruck* kleiner als Null ist.

---

`IFGE  Absolutausdruck`

Liefert Wahr, wenn *Absolutausdruck* größer oder gleich Null ist.

---

`IFLE  Absolutausdruck`

Liefert Wahr, wenn *Absolutausdruck* kleiner oder gleich Null ist.

---

`ELSE`

ELSE kehrt den Wahrheitswert der letzten IF-Direktive um. Beispiel:

```
IFD DEBUG
   ...       ; Teil, der assembliert wird, wenn DEBUG definiert ist
ELSE
   ...       ; Teil, der assembliert wird, wenn DEBUG NICHT definiert
ENDC         ; beende die bedingte Assemblierung
```

Synonym: ELSEIF.

---

`ENDC`

ENDC beendet einen Abschnitt einer IF-Direktive. Falls es noch höhere IF-Direktiven-Verschachtelungen gibt, wird die bedingte Assemblierung aber fortgeführt, z.B.

```
IFEQ  wert        ; Stelle 1
   ...
   IFNE  zahl     ; Stelle 2
   ...
   ENDC           ; beendet IF von Stelle 2
ENDC              ; beendet IF von Stelle 1
```

Synonym: ENDIF.

---

### 3.6.6 Makros

Makros sind Zusammenfassungen mehrerer Quelltextzeilen, die nach ihrer Definition auf Abruf in die Assemblierung eingeschoben werden. Sie können mit Parametern versehen werden, so daß Sie wiederkehrende Sequenzen von Quellzeilen nur einmal angegeben müssen, auch wenn diese geringfügige Unterschiede aufweisen. Ein Makro wird aufgerufen, indem sein Name als Opcode im Opcodefeld der Zeile steht.

Makros können beliebig tief geschachtelt werden und sich selber rekursiv aufrufen. Es gibt die folgenden Direktiven für sie:

---

`Label  MACRO`

Der Assembler definiert *Label* als Makrosymbol und beginnt die Makroaufnahme. *Label* kann Assemblerdirektiven redefinieren, die Erkennung eines Makroaufrufes ist aber im Gegensatz zu den Direktiven abhängig von der Groß- und Kleinschreibung, so daß z.B. RTS immer noch als Prozessorbefehl erkannt wird, wenn ein Makrosymbol namens »Rts« existiert. Normalerweise prüft OMA bei einem Opcode zuerst, ob es eine Direktive ist, und dann, ob es ein Makro ist. Sobald Sie eine Direktive redefiniert haben, arbeitet der Assembler aber prinzipbedingt langsamer! Ab dem Zeitpunkt wird nämlich zuerst geprüft, ob der Opcode ein Makro ist. Für die Liste der nicht redefinierbaren Symbolnamen siehe SET-Direktive.

Innerhalb der Makroaufnahme kann keine erneute Makroaufnahme erfolgen; die Verschachtelung von MACRO-Direktiven ist also unzulässig.

---

`ENDM`

ENDM beendet die aktuelle Makroaufnahme. Der Makro kann von nun an aufgerufen werden. Nach ENDM kann auch eine Aufnahme für einen anderen Makro gestartet werden.

Synonym: ENDMAC.

---

`MEXIT`

MEXIT unterbricht die Makroexpansion der momentanen Hierarchie-Stufe der Makroverschachtelung. Der Assembler sorgt auch dafür, daß die evtl. im Makro benutzte IF-Verschachtelung auf den Stand vor der Expansion dieses Makros gebracht wird.

---



### 3.6.7 Makro-Parameter, -Spezialbefehle und -Beispiele

Wenn einem Makro Parameter mitgegeben werden, stehen sie wie bei Direktiven im Operandenfeld. Die Parameter werden als String in die Makroexpansion gebracht und können Leerzeichen enthalten, wenn der betreffende Parameter in spitze Klammern »<« und »>« oder in eckige »[« und »]« eingefaßt ist. Brauchen Sie »>« oder »]« selber in einem Parameter, können Sie den Parameter entweder mit der jeweils anderen Klammer einfassen oder das Zeichen doppelt angeben (»>>«, »]]«), sofern der Parameter schon mit der entsprechenden Klammer eingefaßt wurde.

Im Makro werden die Parameter mit der Kombination Backslash-Zahl bzw. -Buchstabe angesprochen. »\1« bis »\9« bezeichnet also die Parameter eins bis neun und »\a« bzw. »\A« bis »\z« bzw. »\Z« die Parameter zehn bis fünfunddreißig. Wollen Sie den Backslash selber in der expandierten Zeile erscheinen lassen, geben Sie ihn einfach doppelt an: »\\« erzeugt »\«.

Das Spezialsymbol NARG (kein Case-Unterschied) enthält für jeden aufgerufenen Makro die Anzahl der ihm mitgegebenen Parameter (auch Leerstrings werden mitgezählt).

Es folgen einige Makrobeispiele:

---

**a)**

```
CALL  macro
   IFEQ  NARG-2          ; zwei Parameter?
      move.l \2,a6       ; ja, dann zweiten Parameter nach A6 laden
   ENDC
   IFLE  NARG-2          ; zwei oder kein Parameter?
      jsr    _LVO\1(a6)  ; dann rufe Systemlibrary auf
      MEXIT              ; und beende Expansion
   ENDC
   FAIL  CALL Makrofehler  ; Fehler, wenn mehr als zwei Parameter
      endm
```

Der Aufruf `CALL Delay` expandiert zu `jsr _LVODelay(a6)`. Der Aufruf `CALL Wait,_SysBase` expandiert zu

```
      move.l _SysBase,a6
      jsr    _LVOWait(a6)
```

Der Aufruf `CALL` expandiert zu `jsr _LVO(a6)` und erzeugt höchstwahrscheinlich einen Fehler. Der Aufruf

```
      CALL  1,2,3
```

expandiert zu

```
      move.l 2,a6
      jsr    _LVO1(a6)          (höchstwahrscheinlich mit Fehler)
      FAIL  CALL Makrofehler    (da mehr als zwei Parameter)
```

Der dritte Parameter wird überhaupt nicht verwendet.

---

b)

```
STRING   macro
      dc.b  '\1',0
      endm
```

Der Aufruf STRING <Dies ist ein String> oder STRING [Dies ist ein String] expandiert zu

```
      dc.b  'Dies ist ein String',0
```

---

c)

```
ERZEUGE  macro
wert  SET   \1
   IFNE  wert            ; schon Null erreicht?
      dc.b  0            ; nein, noch ein Nullbyte erzeugen
      ERZEUGE  wert-1    ; und uns selber rekursiv aufrufen
   ENDC
      endm
```

Dieser Makro ruft sich selber rekursiv auf und erzeugt soviele Nullbytes wie im Parameter des ersten Aufrufes angegeben!

Der Aufruf ERZEUGE 3 expandiert zu

```
      dc.b  0
      dc.b  0
      dc.b  0
```

»wert« wird deswegen jedesmal mit SET dem Parameterwert zugewiesen, damit durch den rekursiven Aufruf »\1-1« statt »wert-1« nicht eine überlange Quellzeile expandiert würde (z.B. »300-1-1-1-1-1-1-1...« usw.). »ERZEUGE« kann wegen der beliebig tiefen Makro- und IF-Verschachtelung beliebig viele Nullbytes erzeugen. ERZEUGE 100000 könnte nur an der Speichergröße scheitern. Dieser Makro ist ein etwas akademisches Beispiel; Sie können das gleiche Resultat einfacher erreichen, indem Sie DCB oder die REPT-Direktive verwenden (siehe auch dort), z.B.:



```
   rept 3
   dc.b 0
   endr
```

---

d)

```
Multistring macro
   ifnc '','\1'              ; ist String nichtleer?
      dc.b  '\1',0           ; ja, dann erzeuge ihn
      Multistring \2,\3,\4,\5,\6,\7,\8,\9
   endc
      endm
```

Der Makro kann bis zu zehn Strings in einem Rutsch erzeugen (rekursiver Aufruf). Er bricht ab, sobald der erste Leerstring erscheint, was spätestens nach dem zehnten String der Fall ist, da nicht angegebene Parameter zu Leerstrings expandiert werden. Der Aufruf

```
   Multistring <Der erste>,[der zweite],EinerOhneLeerzeichen
```

expandiert zu

```
      dc.b  'Der erste',0
      dc.b  'der zweite',0
      dc.b  'EinerOhneLeerzeichen',0
```

---

e)

Der Parameter \0 hat eine besondere Stellung. Er wird durch die dem Makroaufruf mitgegebene Größenangabe (B, W, L, S, D, X oder P) ersetzt.

```
Push  macro
      move.\0  \1,-(SP)
      endm
```

Dieser Makro schiebt die angegebenen Werte auf den Stack. Der Aufruf Push.w d0 expandiert zu

```
      move.w   d0,-(SP)
```

Der Aufruf Push.L a4 expandiert zu

```
      move.L   a4,-(SP)
```

Der Aufruf Push d0 expandiert zum fehlerhaften

```
      move.    d0,-(SP)
```

Also muß unser Makro etwas komplizierter werden:

```
Push  macro
   ifc   '','\0'
      move     \1,-(SP)        ; wenn \0 Leerstring ist
   else
      move.\0  \1,-(SP)        ; sonst
   endc
      endm
```

---

**f)**

Der Parameter \@ (Backslash - Kommerzielles At) wird durch eine Zahl ersetzt, die angibt, wie oft bereits Makros expandiert wurden. Diese Zahl ist also für jeden expandierten Makro einzigartig und kann zum Definieren von unverwechselbaren Symbolen verwendet werden. Vor sie wird automatisch ein Unterstrich gesetzt, damit die Labelsyntax korrekt ist, wenn \@ am Zeilenanfang steht.

```
BSTRING  macro
      dc.b  ende\@-*-1      ; ergibt die Länge des Strings
      dc.b  '\1'
ende\@
      endm
```

Dieser Makro definiert einen BCPL-String, der nicht mit einem Nullbyte endet, sondern in seinem ersten Byte die Länge des Strings enthält. Wenn es die 45. Expansion ist, dann expandieren die beiden Zeilen

```
   BSTRING  <Beispiel>
   BSTRING  Beispiel
```

beide identisch zu der Sequenz

```
      dc.b  ende_45-*-1     (Differenz ende_45 zum Stringbeginn ist 8)
      dc.b  'Beispiel'
ende_45
```



---

Der Assembler kennt ferner noch einige Spezial-Makrobefehle, die die Möglichkeiten der Textverarbeitung bei einer Makroexpansion ganz erheblich erweitern. Diese Befehle werden wie Parameter durch Zeichenfolgen ersetzt, und es besteht kein Case-Unterschied.

***VALOF(Symbol)**

wird durch den Dezimalstring des Symbolwertes von *Symbol* ersetzt: *valof(wert) ergibt 65, wenn *wert* den Wert 65 hat.

***LEFT(Text,n)**

ergibt die n linken Zeichen des Strings *Text*: *left(Eiffelturm,6) ergibt Eiffel.

***RIGHT(Text,n)**

ergibt die n rechten Zeichen des Strings *Text*: *right(Eiffelturm,4) ergibt turm.

***MID(Text,n,m)**

ergibt die m Zeichen ab der Position n im String *Text*: *mid(Donaudampfschiffahrt,6,5) ergibt dampf.

***STRLEN(Text)**

wird durch den Dezimalstring der Stringlänge von *Text* ersetzt. *strlen(Assembler) ergibt 9.

Diese Spezialbefehle dürfen nur innerhalb von Makros verwendet werden und können rekursiv geschachtelt werden, wobei Sie jedoch beachten sollten, daß der Stack ausreichend groß ist, da für jede Verschachtelung 512 Bytes benötigt werden. Beispiele für Rekursion sind:

```
\*VALOF(\*LEFT(MeinSymbol,4))  expandiert zu
\*VALOF(Mein)                  und zum Dezimalstring
65                             wenn ">Mein"< den Wert 65 hat
```

---

**g)**

Es soll ein Makro definiert werden, der prüft, ob der mitgegebene Parameter mit »Donau« beginnt, und sonst einen Fehler erzeugt:

```
PRÜFE macro          ; nationale Zeichen sind als Label zulässig!
   IFNE  NARG-1      ; ein Parameter?
      FAIL  falsche Parameter
   ENDC
   IFNC  '\*left(\1,5)','Donau'     ; korrekter Beginn?
      FAIL  falscher Beginn
   ENDC
   endm
```

Der Aufruf `PRÜFE Donaudampfschiffahrt` erzeugt keine Fehlermeldung. Der Aufruf `PRÜFE Editor` erzeugt die Fehlermeldung »FAIL falscher Beginn«. Der Aufruf `PRÜFE` erzeugt »FAIL falsche Parameter« *und* »FAIL falscher Beginn«.

---

h)

Sie möchten (maximal drei) Zeichen im Bereich von »A« bis »Z« jeweils in 5 Bit kodieren und folgendermaßen in ein 16-Bit-Speicherwort packen: `%0aaaaabbbbbccccc`. Dabei ist »aaaaa« der erste Buchstabe usw. Sie benutzen:

```
QUETSCH  MACRO
zeichen1 SET "\*MID(\1,1,1)"-"A"  ; schneide Zeichen aus String
zeichen2 SET "\*MID(\1,2,1)"-"A"  ; das Ergebnis der Zuweisungen
zeichen3 SET "\*MID(\1,3,1)"-"A"  ; ist eine Zahl von 0 bis 25
         dc.w  (zeichen1<<10)|(zeichen2<<5)|zeichen3
         ENDM
```

---

i)

Die folgenden Makros könnten benutzt werden, um einfach und schnell Menüs unter Intuition zu definieren. Beachten Sie dabei, daß der Text nach der MACRO-Direktive als Kommentar angesehen wird und nur der Erläuterung eines Lesers der Makros dient:

```
MENU  MACRO NextMenu,LeftEdge,Width,FirstItem,Name
      dc.l  \1
      dc.w  \2,0,\3,11,MENUENABLED
      dc.l  1$,\4
      dc.w  0,0,0,0
1$    dc.b  '\5',0
      CNOP  0,2
      ENDM

ITEXT MACRO Text
      dc.b  0,1,RP_JAM1
      dc.w  2,1
      dc.l  0,2$,0
2$    dc.b  '\1',0
      CNOP  0,2
      ENDM

ITEM  MACRO NextItem,TopEdge,MutualExclude,Flags,Key,ItemText
      dc.l  \1
      dc.w  0,\2,BREITE,ITEM_HOEHE
      dc.w  \4+ITEMTEXT+ITEMENABLED+HIGHCOMP
      dc.l  \3
```



```
      dc.l  1$,0
      dc.b  \5
      dc.l  0
      dc.w  0
1$    ITEXT <\6>
      ENDM
```

---

Sie finden im Examples-Directory der Originaldiskette weitere Beispiele für Makros, insbesondere beim Quelltext der Library OMA.lib. Auch im Kapitel über die objektorienterte Programmierung7 mit OMA wird intensiv von Makros Gebrauch gemacht, um Übersicht zu erzeugen und Fehler zu vermeiden.

## 3.7 Konvertierung anderer Quelltexte, Inkompatibilitäten

OMA ist durch seine vielen Synonyme und seine umfangreiche Makrounterstützung in der Lage, Quelltexte anderer Assembler ohne größere Änderungen zu assemblieren. Das bekannte Programm *Oktalyzer* wurde z.B. in nur einer halben Stunde auf OMA umgesetzt.

Ein Beispiel: Bei dem Public-Domain-Assembler Asm68k gibt es Direktiven namens CSTRING und PSTRING. CSTRING erzeugt einen String mit einem angehängten Nullbyte und PSTRING erzeugt einen String ohne Nullbyte am Ende, aber mit der Stringlänge als erstem Byte. Die Simulationsmakros können in einem Includefile gelassen werden, das z.B. durch die »-h«-Option beim Assembleraufruf angegeben wird (dann muß der Quelltext nicht geändert werden). Die Makros lauten:

```
CSTRING  macro
      dc.b  '\1',0
      endm

PSTRING  macro
      dc.b  ende\@-*-1,'\1'
ende\@
      endm
```

Wegen OMAs Devpac- und Metacomco-Kompatibilität können deren Quelltexte (beinahe) direkt übernommen werden. Die genauen Unterschiede sind:

### 3.7.1 Inkompatibilitäten zum Metacomco-Assembler Version 11.00

- Der RORG-Ausdruck darf bei OMA nicht kleiner als der Programmzähler sein. Der Metacomco akzeptiert dies, vergißt aber, zwischenzeitlich erzeugte Relokationen u.ä. wieder rückgängig zu machen, und erzeugt u.U. für dieselbe Stelle mehrere Relokationen u.ä.
- ASL <ea> und die übrigen Speicher-Shift-Befehle assembliert OMA nicht mit den Größenangaben B und L, da Speicher-Shifts immer auf 16 Bit Breite erfolgen. Der Metacomco akzeptiert B und L.
- Metacomco akzeptiert einen relativen Ausdruck für MOVEQ, OMA aber nicht.
- Metacomco expandiert für den \@-Makroparameter die Form .nnn mit einem einleitenden Punkt, OMA aber _nnn mit einem einleitenden Unterstrich, da der Punkt am Anfang eines Labels normalerweise ein lokales Label einleitet.



### 3.7.2 Inkompatibilitäten zum Devpac-Assembler Version 2, Oktober 1988

- Die Aufrufsyntax des Devpac ist völlig anders.
- Die reservierten Symbol __LK und __G2 existieren nicht, da OMA nicht der Devpac-Assembler ist!
- Die Operatoren »=«, »<« und »>« existieren nicht, und »~« hat bei OMA die gleiche Priorität wie monadisches Plus bzw. Minus.
- R0 bis R15 werden NICHT als Synonyme für D0 bis D7 und A0 bis A7 erkannt. Dies widerspräche dem Motorola-Standard. Brauchen Sie das trotzdem, definieren Sie die Registersymbole einfach per EQUR.
- \W und \L werden NICHT als Synonyme für .W und .L zur Erzwingung der Adressierungsarten »Absolut-Lang« bzw. »-Kurz« erkannt. Diese Syntax widerspricht dem Motorola-Standard.
- Der unmittelbare MOVEQ-Wert darf nicht im Bereich 128-255 liegen. Wollen Sie trotzdem eine solche vorzeichenlose Byte-Konstante »xx« laden, benutzen Sie MOVEQ #xx-256,Dn. Beachten Sie aber, daß dann die Bits 8-31 des Datenregisters gesetzt sind.
- ORG, IIF und XREF.L existieren nicht.
- COMMENT, FORMAT und SUBTTL haben keine Wirkung.
- LIST bei OMA kennt die Parameter »+« und »-« nicht.
- LLEN darf bei OMA minimal 60 sein, Devpac geht herunter bis 38.
- INCBIN legt bei OMA die Bytes nicht automatisch auf Wortgrenze und füllt nach dem Inkludieren nicht auf Wortgrenze auf.

Bei der OPT-Direktive gibt es erhebliche Unterschiede. Im folgenden wird zuerst die Devpac-Konvention erläutert, dann kommen die Verhältnisse unter OMA.

- A: Automatische PC-relativ-Konvertierung, A+ an, A- aus (default). Bei Bei OMA unzulässig, da defaultmäßig immer optimiert wird.
- C: Case-Unterschied, C+ kein Unterschied, C- Unterschied (default). Zusätzlich kann gleich nach dem C dezimal angegeben werden, wieviele Stellen von Symbolen signifikant sind. Bei OMA ist die Signifikanz immer 255 Zeichen, C+ und C- werden aber verstanden und entsprechen der »-c c«-Option. Allerdings ist Devpacs C+ (kein Unterschied) bei OMA C-, und OMAs C+ schaltet die Unterscheidung **an**.
- L: Linkbares Objektfile, L+ an, L- ausführbar (default). Bei Nichtbenutzung dieser Option entscheidet OMA selber zwischen ausführbar und linkbar. Ferner meldet OMA Fehler bei der XREF/XDEF-Direktive, wenn L- gewählt wurde.
- M: Makroexpansion im Listing, M+ an, M- aus (default). Bei OMA ist M+ voreingestellt.

- O: Optimierungen (defaultmäßig aus), O1 Short Branches, O2 0(An)-Optimierung, O+ alles an, O- alles aus, OW Warnings beim Optimieren an/aus. Da OMA defaultmäßig immer optimiert, gibt es O1, O2, OW nicht. O+/O- entsprechen OPTIMIZE/NOOPTIM.
- S: Symboltabelle im Listing, S+ an (default), S- aus. Bei OMA ist S- voreingestellt.
- T: Typprüfung, T+ an (default), T- aus. Bei OMA unzulässig, der Typ wird immer geprüft.
- X: Es gelangen nur exportierte Symbole in die Debug-Information. X+ an, X- aus (default). Bei OMA nur gültig, falls nicht »-c s« und nicht »OPT D+« verwendet wurden.

Makros:

- Wenn kein Size Specifier angegeben ist, entspricht der Parameter \0 bei OMA dem Leerstring und nicht W.
- Die Spezialkonversion »\<symbol>« bzw. »\$<symbol>« existiert nicht. Für »\<symbol>« kennt OMA den speziellen Makrobefehl *VALOF.
- Ein Makroaufruf über mehrere Zeilen ist unzulässig.
- Makronamen können mit anderen Symbolen kollidieren.



## 3.8 Beginn einer Sektion

Zu Beginn eines Assemblerlaufes ist die namenlose Code-Sektion eingeschaltet. Die Reihenfolge der Sektionen im Objektfile hängt aber nicht von der Reihenfolge der eingeschalteten Sektionen ab, sondern von der Reihenfolge der **aktivierten** Sektionen. Deswegen kann die zuerst eingeschaltete Code-Sektion überhaupt nicht im Objektfile erscheinen. Die zuerst gewünschte Sektion sollte daher immer als eine der ersten Direktiven angegeben werden.

Die meisten Direktiven aktivieren die gerade eingeschaltete Sektion. Die folgenden Direktiven aktivieren die Sektion **nicht**:

**IF-Direktiven, ELSE, ENDC, RS-Direktiven, ORG, SECTION, CODE, DATA, BSS, XDEF, XREF, OPTIMIZE, NOOPTIM, INCLUDE, INCDIR, EQU, SET, EQUR, SETR, REG, SETREG, FEQU, FSET, REPT, ENDR, MACRO, ENDM, MEXIT, NOOBJ, OUTPUT, END, OPT, LLEN, PLEN, LIST, NOLIST, CLIST, NOCLIST, MLIST, NOMLIST, FORMAT, NOFORMAT, PAGE, NOPAGE, TTL, SPC, MASK2, FAIL, IDNT, COMMENT, SUBTTL**

Darüberhinaus wird eine Sektion auch aktiviert, wenn in ihr ein relatives Symbol deklariert wird. Beispiele:

1. Die Data-Sektion wird als erste aktiviert, da XDEF nicht aktiviert:

```
   CODE
   XDEF  symbol
   DATA
   NOP              ; aktiviert
```

2. Die Code-Sektion wird trotz EQU nicht als erste aktiviert:

```
   CODE
wert  EQU   1       ; aktiviert nicht
   DATA
   NOP              ; aktiviert
```

3. Das relative Symbol aktiviert die Sektion:

```
   SECTION Parser,CODE
Parserstart         ; aktiviert
```

4. **Achtung:** Hier beginnt die Programmausführung in der Bss-Sektion, da diese durch DS vor der Code-Sektion aktiviert wurde! Auf solche Fehler sollten Sie *unbedingt* achten!

```
   bss
variable ds.w  1    ; aktiviert
   code
   nop
```

## 3.9 Optimierungen und Konvertierungen

OMA trägt seinen Namen nicht umsonst, denn er optimiert effizient an allen möglichen Stellen, allerdings nur bei eingeschaltetem Optimierer und nur dort, wo es ohne Risiko möglich ist. Im Mehrpass-Modus werden auch Ausdrücke mit Vorwärtsreferenzen optimiert, so daß auch z.B. Vorwärtssprünge in ihre Kurzform überführt werden können und der kürzeste Code generiert wird.

Sie können Optimierungen verhindern (z.B. bei selbstmodifizierendem Code, den ich persönlich aus Kompatibilitätsgründen ablehne, weil solche Software nicht auf MC68020 und höheren Prozessoren läuft), indem Sie z.B. »bra.w« o.ä. schreiben. Siehe auch Kapitel 3.5 über die Adressierungsarten.

Folgende Optimierungen finden bei den Adressierungsarten statt:

a(An) mit a=0 wird zu (An) optimiert.

Absolut lang wird zu Absolut kurz oder zu PC-relativ optimiert.

Bei den zusätzlichen Adressierungsarten der 32-Bit-Prozessoren wird die Form erzeugt, die den kürzesten Code ermöglicht.

Weitere Optimierungen bzw. Konvertierungen sind:

ADD #a,<ea> mit 1 <= a <= 8 wird zu ADDQ optimiert.
ADD #a,<ea> mit -8 <= a <= -1 wird zu SUBQ optimiert.
ADD wird zu ADDA bzw. ADDI konvertiert.

ADDA #a,An mit 1 <= a <= 8 wird zu ADDQ optimiert.
ADDA #a,An mit -8 <= a <= -1 wird zu SUBQ optimiert.
ADDA.L #a,An mit -32768 <= a <= 32767 wird zu »ADDA.W #a,An« optimiert.

ADDI #a,<ea> mit 1 <= a <= 8 wird zu ADDQ optimiert.
ADDI #a,<ea> mit -8 <= a <= -1 wird zu SUBQ optimiert.

AND wird zu ANDI konvertiert.

BHS und BLO sind Synonyme von BCC und BCS.
Branch-Befehle werden in die Kurzform optimiert. Sind sie dann Null, werden sie durch NOP ersetzt und bei BSR ein Fehler gemeldet (Weglassen von Nullbranches würde im Mehrpass-Modus zum »Gummiband-Effekt« führen: Der Assembler macht immer wieder neue Passes und terminiert nicht).
Ein BSR mit Nulldistanz zur folgenden Instruktion muß als BSR.W geschrieben werden, um einen Fehler zu verhindern.
Ungerade Branches erzeugen ein Warning.

CMP #a,<ea> mit a=0 wird zu TST optimiert.
CMP wird zu CMPA, CMPI oder CMPM konvertiert.

CMPI #a,<ea> mit a=0 wird zu TST optimiert.

CLR.L Dn wird zu »MOVEQ #0,Dn« optimiert.



DBHS und DBLO sind Synonyme von DBCC und DBCS.
DBRA ist ein Synonym zu DBF.
Ungerade Branches erzeugen ein Warning.
Alle Divisionsbefehle mit unmittelbarem Nulloperanden melden ein Warning: Division durch Null.

EOR wird zu EORI konvertiert.

JMP wird zu BRA optimiert.

JSR wird zu BSR optimiert.

MOVE.x #a,<ea> mit a=0 wird zu »CLR.x <ea>«, »SUB.L An,An« oder »MOVEQ #0,Dn« optimiert.

MOVE.L #a,Dn mit -128 <= a <= 127 wird zu »MOVEQ #a,Dn« optimiert.

MOVEA.x #a,An mit a=0 wird zu »SUBA.L An,An« optimiert.

Alle Multiplikationsbefehle mit unmittelbarem Nulloperanden melden ein Warning: Multiplikation ergibt Null.

OR wird zu ORI konvertiert.

SHS und SLO sind Synonyme von SCC und SCS.

SUB #a,<ea> mit 1 <= a <= 8 wird zu SUBQ optimiert.
SUB #a,<ea> mit -8 <= a <= -1 wird zu ADDQ optimiert.
SUB wird zu SUBA bzw. SUBI konvertiert.

SUBA #a,An mit 1 <= a <= 8 wird zu SUBQ optimiert.
SUBA #a,An mit -8 <= a <= -1 wird zu ADDQ optimiert.
SUBA.L #a,An mit -32768 <= a <= 32767 wird zu »SUBA.W #a,An« optimiert.

SUBI #a,<ea> mit 1 <= a <= 8 wird zu SUBQ optimiert.
SUBI #a,<ea> mit -8 <= a <= -1 wird zu ADDQ optimiert.

TRAPHS, THS und TRAPLO, TLO sind Synonyme von TRAPCC, TCC und TRAPCS, TCS.

## 3.10 Fehlermeldungen

Der Assembler bietet ausführliche und präzise Fehlermeldungen, die die Fehlerkorrektur erleichtern und in vier verschiedene Kategorien eingeteilt werden können:

1. Fehler kleiner 100 sind fatale Fehler, die gleich beim Assemblerstart auftauchen.
2. Fehler von 100 bis 399 sind normale Fehler beim Assemblieren von Quelltextzeilen.
3. »Fehler« von 400 bis 499 sind Warnings, die nur auf außergewöhnliche Zustände hinweisen und keine echten Fehler sind. Der Quelltext wird höchstwahrscheinlich korrekt übersetzt, aber Sie sollten solche Zeilen kontrollieren, um sicherzustellen, daß alles nach Wunsch lief. Ein Beispiel ist die Meldung des Assemblers, daß er eine Prozessorinstruktion auf eine gerade Adresse legt. Ein Label der vorigen Zeile könnte dann ungerade sein und eine Adreß-Exception des Prozessors beim Ansprung dieses Labels auslösen.
4. Fehler größer als 900 sind fatale Fehler, die erst während des Assemblerlaufes auftreten und die Fortsetzung der Assemblierung unmöglich machen. Ein Beipiel ist eine interne Division durch Null.

Einige Fehler fordern Sie auf, den Fehler beim Hersteller von OMA zu melden. In dem Fall sollten Sie dies unverzüglich tun, da diese Fehler beim normalen Ablauf des Assemblers nicht vorkommen sollten und evtl. einen Programmfehler in OMA anzeigen.

Die Fehlermeldungen selber werden hier nicht aufgeführt. Zum einen sind sie präzise genug, um sofort verstanden zu werden, zum anderen wäre es enorm viel Platz- und Zeitverschwendung, sämtliche über 300 Fehler und Warnings mit den Umständen aufzuführen, unter denen sie entstehen.

Falls Sie den Assembler in Batch-Dateien verwenden wollen, sollten Sie folgendes beachten:

- Tritt ein fataler Fehler auf, wird als Fehlercode die Nummer des Fehlers zurückgegeben (im Bereich 21-99 bzw. 900-999).
- Tritt sonst mindestens ein einfacher Fehler im Bereich 100-399 auf, wird als Fehlercode 10 zurückgegeben.
- Tritt sonst mindestens ein Warning auf (Bereich 400-499), wird als Fehlercode 5 zurückgegeben.
- Sonst wird als Fehlercode 0 zurückgegeben, d.h., es gab weder Fehler noch Warnings.



## 3.11 Verkürzte Adressierung

In vielen Programmen werden häufig Variablen angesprochen, die zu einer Bss-Sektion gehören. Dies führt z.B. zu folgendem Code:

```
   CODE
   move  d0,wert1     ; kein PC-relativ, da Zieladressierung
   move  wert2,d0     ; kein PC-relativ, da andere Sektion

   BSS
wert1 ds 1
wert2 ds 2
```

Jeder MOVE-Befehl braucht hier drei Worte. Nun kann die Codegröße und die Ausführungszeit gesenkt werden, wenn es möglich ist, die Adressierung von »Absolut Lang« in »Adreßregister indirekt mit Offset« zu wandeln. Dann bräuchte jeder MOVE-Befehl nur noch zwei Worte und wäre auf einer MC68000 vier Taktzyklen schneller. Eine Möglichkeit besteht in der Benutzung des Basereg-Modus (siehe BASEREG-Direktive), aber dies löst nicht das folgende Problem bei Programmen, die in mehrere Module zerlegt sind:

File a.asm:

```
   CODE
   XREF  wert,text
   move  d0,wert
   lea   text,a0
   rts
   END
```

File b.asm:

```
   BSS
   XDEF  wert
basis
   ds.l  1
wert
   ds.l  1
   END
```

File c.asm:

```
   CODE
   XDEF  text
text  ds.b  'Beispiel'
   END
```

a.asm kann nicht wissen, daß »wert« als extern geltendes Symbol verkürzt adressiert werden könnte. Nun dürfen aber auch nicht einfach alle externen Werte verkürzt adressiert werden, da sonst auf »text« falsch zugegriffen würde. Es gibt leider keine andere Lösung als die folgende:

File a.asm:

```
   CODE
   XREF  basis,wert,text
   lea   basis+32768,a5  ; initialisiere den Basiszeiger (wie BASEREG)
                         ; Die Initialisierung MUß erfolgen!
   move  d0,wert(a5)     ; Verkürzung muß (leider) extra angegeben werden,
                         ; es entsteht eine 16-Bit-Referenz
   lea   text,a0         ; funktioniert fehlerlos
   rts
   END
```

File b.asm muß zusätzlich »basis« exportieren:

```
   BSS
   XDEF  basis,wert
basis
   ds.l  1
wert
   ds.l  1
   END
```

File c.asm:

```
   CODE
   XDEF  text
text  ds.b  'Beispiel'
   END
```

Nun tritt aber noch ein Problem auf: Normale Linker spielen beim Linken der zugehörigen Objektfiles a.obj, b.obj und c.obj nicht mit, da eine 16-Bit-Referenz mit einem Symbol gelöst werden soll, das nicht zur gleichen Sektion gehört. Dies führt zu einer Fehlermeldung.

**Aber:** Mit der SHORT-Option des im Assemblerpaket enthaltenen Linkers DLink werden die Objektfiles korrekt gelinkt, falls der resultierende 16-Bit-Wert nicht außerhalb seines Bereiches gerät und das Symbol zu einer Bss-Sektion gehört. Die Referenz in diesem Beispiel ergibt `wert-32768 = 4-32768 = -32764` für den Offset, also lautet der Befehl im fertigen Programm

```
MOVE D0,-32764(A5)
```



Der Linker sorgt automatisch für den zu berücksichtigenden Offset von -32768.

Der Short-Modus funktioniert auch, wenn mehrere Objektfiles Teile ein und derselben Bss-Sektion enthalten, solange die Basisadresse in dem Objektfile definiert wird, das als erstes von diesen Objektfiles in der Aufrufzeile steht. Für das folgende Beispiel lautet der Linkeraufruf `DLink a.obj b.obj c.obj to prg SHORT`. File a.asm:

```
   CODE
   XREF  basis,wert1,wert2
   lea   basis+32768,a5     ; Initialisierung des Basiszeigers
   move  d0,wert1(a5)       ; wird zu MOVE D0,-32764(A5)
   move  d0,wert2(a5)       ; wird zu MOVE D0,-32760(A5)
   rts
   END
```

File b.asm:

```
   BSS
   XDEF  basis,wert1
basis
   ds.l  1
wert1
   ds.l  1
   END
```

File c.asm:

```
   BSS
   XDEF  wert2
wert2
   ds.l  1
   END
```

Zur einfachen Behandlung dieser Adressierung sollte für den Basiszeiger eine Zuweisung der Form

```
BP    EQUR  A5        ; oder anderer Name
```

getroffen werden, damit Befehle z.B. als »MOVE D0,wert1(BP)« geschrieben werden können. Dadurch können Verwechselungen mit anderen Adreßregistern vermieden werden. Um das Beschreiben des Basisregisters zu kontrollieren, kann der Basereg-Modus eingeschaltet werden, denn dann wird ein Warning gemeldet, wenn das Basisregister das Ziel eines Prozessorbefehls ist!

Hier möchte ich auch auf das entsprechende Kapitel im Linker-Handbuch verweisen.

## 3.12 Das Listingfile

Das Listingfile des Assemblers enthält vielfältige Information über den Assemblerlauf und kann mit vielen Optionen beeinflußt werden. Hier sei auf die Listing-Direktiven des Kapitels 3.6 verwiesen. Das Listingfile erzeugen Sie mit der »-l«- oder »-c l«-Startoption, siehe Kapitel 3.2. Zum besseren Verständnis sollten Sie mit dem Listingfile experimentieren, während Sie dieses Kapitel lesen.

Der Assembler erzeugt im Listing in regelmäßigen Abständen einen Seitenvorschub, der in der ersten Zeile den String der TTL-Direktive und die Seitennummer und in der dritten Zeile die Felderaufteilung (auch für die Cross-Referenz und die Symboltabelle) enthält. Dieser Seitenvorschub ist vorerst eingeschaltet und kann durch die »-c p«-Option und die Direktiven NOPAGE, PAGE und TTL beeinflußt werden.

Falls Fehler oder Warnings auftraten, erzeugt der Assembler noch eine Fehlerliste, in der diese nach der Reihenfolge des Auftretens enthalten sind. Die Fehlerliste wird mit einem Seitenvorschub vom Listing abgegrenzt, falls der Seitenvorschub nicht abgeschaltet ist.

Nach dem Listing wird eine Cross-Referenz oder eine Symboltabelle erzeugt, sofern Sie die »-c x«- bzw. »-c t«-Startoption benutzt haben. Wird beides gleichzeitig verlangt, erscheint nur die Cross-Referenz, da sie wie eine Symboltabelle ist, aber Verweise enthält, wo jedes einzelne Symbol benutzt wurde. Cross-Referenz und Symboltabelle werden *immer* vom Listing bzw. der Fehlerliste durch einen Seitenvorschub abgegrenzt. Ist der Seitenvorschub eingeschaltet, wird er regelmäßig auch in die Cross-Referenz bzw. Symboltabelle eingeschoben.

### 3.12.1 Das Listing

Das generelle Format einer Listingzeile lautet:

```
ZEILE     PC          CODE        ZEILE    QUELLZEILE
AAAAAA  $xxxxxxxx: xxxxXXXXxxxxXXXX Ezzzzzz+ qqq...
```

*ZEILE* ist die gesamte Nummer der Zeile im Assemblerlauf. Der Assembler numeriert *alle* assemblierten Zeilen, also auch solche, die z.B. durch eine Makroexpansion erzeugt wurden.

*PC* enthält den aktuellen Programmzähler als Hexadezimalzahl und endet mit einem Doppelpunkt. Dieses Feld kann leer sein (s.u.).

Das *CODE*-Feld enthält in der Regel die durch Prozessorinstruktionen oder Direktiven erzeugten Bytes. Falls nötig, werden für eine Quellzeile noch mehrere Zeilen zusätzlich erzeugt, die ausschließlich das *CODE*-Feld enthalten, damit alle erzeugten Bytes dokumentiert werden.

Das rechte *ZEILE*-Feld enthält die Nummer der Zeile innerhalb des Quellfiles und wird durch ein »E« eingeleitet, wenn die Zeile einen Fehler enthält. Das »+« erscheint, wenn die Zeile durch eine Makroexpansion entstanden ist, also aus einem Makro stammt.



Schließlich wird die Quellzeile im *QUELLZEILE*-Feld untergebracht.

Für spezielle Direktiven enthält das *CODE*-Feld andere Information als die erzeugten Bytes. Im folgenden sind die einzelnen Ausgaben für diese Direktiven beschrieben:

**OFFSET:** Der Anfangswert des Offsetzählers wird mit einem Gleichheitszeichen ausgegeben.

**EQU:** Es erscheint ein Gleichheitszeichen. Wenn der Wert absolut oder relativ ist, folgt er als Hexadezimalzahl. Ist er relativ, folgt noch ein einfaches Hochkomma. Ist der Wert importiert, wird dies als Text kenntlich gemacht.

**SET:** Die Ausgabe ist die gleiche wie bei EQU mit dem Unterschied, daß statt des Gleichheitszeichens die Kombination -> erscheint.

**EQUR:** Es erscheint der Text »= Register Xn«, wobei »Xn« durch das entsprechende Daten- oder Adreßregister ersetzt wird.

**SETR:** Es besteht der gleiche Unterschied wie zwischen EQU und SET.

**REG:** Der Assembler erzeugt »= Maske xxxx«, wobei »xxxx« durch die Maske als 16-Bit-Hexadezimalzahl ersetzt wird.

**SETREG:** Es besteht der gleiche Unterschied wie zwischen EQU und SET.

**DCB:** Der Assembler erzeugt »= xxxxxxxx Bytes«, wobei »xxxxxxxx« durch die Anzahl der erzeugten Bytes als Hexadezimalzahl ersetzt wird.

**DS:** Normalerweise ist die Ausgabe dieselbe wie bei der DCB-Direktive. Im Offsetabschnitt wird der Offsetzähler mit einem Gleichheitszeichen ausgegeben.

**RS:** Der RS-Zähler wird mit einem Gleichheitszeichen ausgegeben.

**INCBIN:** Hier fehlt das Dokumentieren der eingelesenen Bytes. Dies ist sinnvoll, da ein INCBIN-File beliebig groß sein darf. Entsprechend viele Zeilen würden im Listing benötigt, um alle Bytes auszugeben, so daß das Listingfile größer als der ursprüngliche Quelltext würde.

**IF:** Bei den IF-Direktiven erscheint »Wahr«, wenn die Bedingung wahr ist. Sonst erscheint »Falsch«.

**ELSE:** Die Ausgabe ist dieselbe wie bei den IF-Direktiven, bezieht sich aber auf den umgekehrten Wahrheitsgehalt.

### 3.12.2 Cross-Referenz und Symboltabelle

Wie oben schon erwähnt, ist die Cross-Referenz im Prinzip eine Liste mit allen im Assemblerlauf vorgekommenen Symbolen, die für jedes Symbol neben der Zeilennummer der Definitionszeile auch alle Zeilennummern der Zeilen enthält, in denen das jeweilige Symbol auftaucht. Die Cross-Referenz muß mit der »-c x«-Startoption explizit von Ihnen angefordert werden. Das generelle Format einer Cross-Referenz-Zeile lautet:

```
SYMBOL                              ZEILE  WERT       REFERENZEN
12345678901234567890123456789012 zzzzzz xxxxxxxx'  zzzzzz
```

Das *SYMBOL*-Feld enthält den Namen des Symbols. Falls dieser länger als 32 Zeichen ist, wird er komplett ausgegeben, und die übrigen Felder bekommen eine eigene Zeile.

Das *ZEILE*-Feld enthält die Gesamt-Zeilennummer der Zeile, in der das Symbol definiert wurde (**nicht** die Nummer innerhalb des Quellfiles!).

Das *WERT*-Feld enthält den Wert bzw. die Art des Symboles. Ist es ein relativer Wert, wird noch ein einfaches Hochkomma angehängt. Das Symbol kann von den folgenden Symboltypen sein:

- Absolut
- Relativ
- Importiert
- Fließkommasymbol
- Registersymbol
- Registermaske
- Makro
- Undefiniertes Symbol (z.B. in ausgeschlossenem IF-Abschnitt)
- Mehrfach definiertes Symbol (d.h., es gab Fehler beim Assemblieren)

*REFERENZEN* enthält dann die Gesamt-Zeilennummern (**nicht** die Nummer innerhalb des Quellfiles!), in denen das Symbol benutzt wurde. Es werden so viele Nummern ausgegeben wie nötig sind, um die mit LLEN eingestellte maximale Zeilenlänge zu erreichen. Wenn nötig, werden weitere Zeilen erzeugt, die die Zeilennummern enthalten.

Falls Ihnen die Cross-Referenz zu groß ist, Sie aber dennoch einige Symbolwerte wissen möchten, dann verlangen Sie per ≫-c t≪-Startoption die Symboltabelle. Bis auf das Fehlen des *REFERENZEN*-Feldes (das evtl. sehr viel Platz beanspruchen kann), ist die Symboltabelle mit der Cross-Referenz identisch.



## 3.13 Vorassemblierte Includefiles

OMA bietet die Möglichkeit, Includefiles vorzuassemblieren und Zeit beim späteren Assemblerlauf zu sparen. Ein File namens prog.asm enthalte beispielsweise die folgende Sequenz:

```
include  "exec/types.i"
include  "intuition/intuition.i"
```

Es werden sehr viele Konstanten und Makros durch die Includefiles gelesen, da intuition.i noch selber etliche Includefiles holt. Für jeden Assemblerlauf heißt dies, daß jedesmal dieselben Werte definiert werden und dieselben Zeilen zu lesen und zu zerlegen sind. Diese Arbeit läßt sich abkürzen.

Assemblieren Sie die obige Sequenz z.B. als File file.asm durch

```
OMA file.asm -c n -t Vorassembliert
```

Sie erhalten dann ein vorassembliertes File namens Vorassembliert, das alle mit EQU, SET, FEQU, FSET, REG, SETREG, EQUR, SETR und MACRO definierten Konstanten und Makros aus den Includefiles enthält (da END fehlt, werden Sie ein Warning bekommen; ≫-c n≪ wird benutzt, damit kein Objektfile erzeugt wird). Entfernen Sie die obige Include-Sequenz aus dem ursprünglichen Quelltext prog.asm, damit die Includefiles nicht mehr gelesen werden, und assemblieren Sie per

```
OMA prog.asm -o prog -p Vorassembliert
```

Der Assembler liest die Konstanten und Makros aus Vorassembliert sehr schnell ein und assembliert den übrigen Quelltext prog.asm. Die zum Assemblieren benötigte Zeit ist im Vergleich zum Lesen der Includefiles stark gesunken. Da OMA in Bezug auf die vorassemblierten Files **sehr** empfindlich ist, sei die folgende Warnung angeführt:

**ACHTEN SIE DARAUF, DAß DIE VORASSEMBLIERTEN INCLUDEFILES VON OMA ERZEUGT UND NICHT IN IRGENDEINER FORM VERÄNDERT WURDEN. SONST WIRD DER ASSEMBLER SEHR GIFTIG REAGIEREN: MIT EINEM SYSTEMABSTURZ.**

Sofern Sie sichere Datenträger verwenden, kann aber so gut wie nichts passieren. Selbst wenn Sie echte Includefiles als vorassemblierte benutzen wollten, würde OMA dies rechtzeitig bemerken und melden. Vorassemblierte Includefiles können auch mittels der Direktive INCPRE gelesen werden.

## 3.14 Source-Level-Debugging

Source-Level-Debugging bedeutet, daß der Quelltext eines Programmes genau so im Debugger verfügbar ist, wie er in der Quelldatei steht. OMA unterstützt eine solch hochbequeme Form des Debuggens. Dabei kann ausgewählt werden, welche Teile als Source-Level-Information vorliegen sollen und welche nicht.

Wenn die Startoption »-c q« angegeben ist, wird der Assembler Quelltextzeilen in einem speziellen Format in einen Debughunk abspeichern. Normalerweise gelangen alle Zeilen außer Leerzeilen in diesen Hunk, die auch in das Listingfile gelangen würden. Wenn gleichzeitig ein Listingfile erzeugt wird, gelangen auch die Leerzeilen in den Debughunk. Dabei ist es dann möglich, die Steuerung über die Listingdirektiven LIST, NOLIST, CLIST usw. auszunutzen, um selektiv Quelltextteile zum Source-Level-Debugging in den Debughunk zu schreiben. Per Makros expandierte Quellzeilen können auch vollständig geschrieben werden. Damit stehen dem Debugger DBug für echtes Source-Level-Debugging alle notwendigen Informationen zur Verfügung!

Module, die ohne Source-Level-Information erzeugt werden, haben im Debugger einfach keine Darstellung in der Source-Level-Ebene.

Leider ist der Debughunk durch Commodore nur schwach genormt, daher können andere Debugger bzw. Linker als DBug bzw. DLink nichts mit den Daten anfangen. Daher sollten Sie unbedingt mit DLink linken. Zudem kann im Extremfall (wenn auch die Includefiles u.ä. in den Debughunk gelangen) der Dateiumfang eines Objektfiles um mehr als das zehnfache zunehmen. Daher sollten nur wenige Programmteile auf voller Source-Level-Ebene betrieben werden bzw. es sollte genug Speicher zur Verfügung stehen.



## 3.15 32-Bit-Prozessoren und 32-Bit-Probleme

Da OMA die 32-Bit-Prozessoren unterstützt, ergeben sich kleinere Probleme gegenüber den 16-Bit-Prozessoren. Sie sollten immer die Form »bra.w« statt »bra.l« verwenden, da der lange Branch im 32-Bit-Modus als 32-Bit-Befehl assembliert wird. Die 16-Bit-Prozessoren interpretieren diesen Befehl falsch und geraten auf eine ungerade Befehlsadresse. Daher erhalten Sie im 16-Bit-Modus bei »bra.l« ein entsprechendes Warning. Beachten Sie ferner, daß jetzt mehr Symbolnamen reserviert sind, um die zusätzlichen Prozessorregister zu unterstützen.

Ferner ist es ein Problem, daß externe, relative 32-Bit-Referenzen durch Amiga-DOS nicht aufgelöst werden können, da entsprechende Hunktypen nicht definiert wurden. Wenn z.B. »that« importiert ist, sind folgende Befehle bzw. Adressierungsarten nicht möglich (dies betrifft nur die 32-Bit-Prozessoren), und OMA meldet einen Fehler:

```
bra.l  that
fbeq.l that
move   (that,pc),d0
move   (that,pc,d1),d0
move   ([that,pc],d1),d0
move   ([that,pc,d1]),d0
```

Sofern Sie Programme für den MC68040-Prozessor schreiben wollen, sollten Sie beachten, daß bei diesem der Cache ganz anders als bei den anderen 32-Bit-Prozessoren angesteuert wird und die PMMU-Befehle eine völlig unterschiedliche Kodierung haben. PMMU-Befehle der MC68030 erzeugen auf dem MC68040 eine Illegal-Opcode-Exception und umgekehrt. Achten Sie also darauf, welchen Prozessor Sie für die Assemblierung bei PMMU-Befehlen ausgewählt haben, und daß Sie ab System 2.0 für Cache-Behandlung die entsprechenden Exec-Routinen benutzen.

# Kapitel 4

# Symbolischer Debugger DBug V1.1

## 4.1 Einführung

DBug ist ein Source-Level-Multiwindow-Debugger für die Commodore-Amiga-Familie. Sein Konzept umfaßt eine durchgängige Bedienung des Programmes durch Maus und Tastatur nebst vollständig symbolischem Debuggen von beliebigen Tasks, die vom CLI oder der Workbench gestartet werden oder auch schon länger im System laufen. DBug läuft auf allen Amiga-Modellen und in Verbindung mit allen Prozessoren der Motorola-68000-Familie sowie der Fließkomma-Coprozessoren. Befehle und Eigenschaften für die höheren Prozessoren werden ebenso automatisch erkannt, verarbeitet und unterstützt wie die neuen Betriebssysteme ab 2.0. In diesem Handbuch sollen Kickstart 1.3 und Kickstart 1.2 gleichbedeutend sein.

Da DBug ausgiebig die Intuition-Fenstertechnik gebraucht, können Sie mit ihm keine Programme debuggen, die das System ausschalten oder andere unsaubere »Tricks« anwenden. Die Anzahl möglicher Fenster selbst gleichen Typs ist nur durch den Speicher begrenzt. Es ist also kein Problem, z.B. 20 Speicherfenster gleichzeitig offen zu haben. Jedes Fenster besitzt aber seine individuelle Einstellung; z.B. beeinflußt eine Änderung der Sortierung des einen Symbolfensters nicht die Sortierung eines anderen.

Hier noch eine kleine Zusammenfassung der Leistungsmerkmale:

- symbolischer Multiwindow-Debugger mit Unterstützung aller Prozessoren
- Source-Level-Unterstützung bietet Abarbeitung direkt im Quelltext
- für alle Amiga-Modelle und alle Betriebssysteme ab Kickstart 1.2
- Debuggen beliebiger Tasks oder Programme
- Einzelschrittmodus, schnelles Ausführen von Unterprogrammen

- Breakpoints, Speicher-Breakpoints, zeitweilig ausschaltbar
- Anzeige beliebiger Speicherbereiche in verschiedenen Formaten, integrierter Assembler und Disassembler für alle Prozessoren
- History-Funktion mit beliebiger Tiefe
- Logfile zum Protokollieren
- fontsensitiv: verwendet den eingestellten System-Defaultfont

**Starten des Debuggers**

Sie starten den Debugger vom CLI, einer Shell oder der Workbench. Für den Workbench-Start klicken Sie einfach sein Icon doppelt an. Vom CLI aus geben Sie den Namen DBug als Befehl ein und drücken Return, DBug startet sich dann intern als selbständige Task und läßt das CLI unabhängig weiterlaufen. Sie können beim CLI-Start gleich ein Programm zum Debuggen anwählen, indem Sie seine CLI-Startzeile an den Debuggernamen anhängen. Dieses Programm wird dann sofort eingeladen und zum Debuggen vorbereitet. Dazu sollten Sie den Programmnamen allerdings so weit angeben, daß der Debugger das Programmfile finden und die Symbole laden kann. Sofern Sie z.B. nicht im C:-Directory sind, reicht der Name filenote nicht aus, Sie müssen dann C:filenote angeben.

Zum Start von der Workbench klicken Sie das Icon des Debuggers wie gewohnt doppelt an. Wollen Sie gleichzeitig schon ein Programm zum Debuggen auswählen, dann verwenden Sie die erweiterte Auswahl wie üblich.

Vor Öffnen des Standardfensters versucht der Debugger, aus dem aktuellen Directory ein File namens DBug.prefs zu lesen. Vom Debugger erzeugt, kann dieses die Fenstereinstellung der letzten Debug-Sitzung enthalten, und DBug öffnet die Fenster exakt so, wie Sie sie abgespeichert haben (siehe auch Window-Menü). Da Amiga-DOS relokatible Programme verwendet, bleibt Ihnen allerdings nicht erspart, z.B. Speicherfenster auf die richtige Startadresse zu setzen. Gibt es kein File namens DBug.prefs oder stimmt dessen Prüfsumme nicht, dann öffnet sich einfach ein normales Statusfenster in der linken oberen Ecke.

DBug kann zwar resident gehalten werden, und Sie können DBug mehrfach nebeneinander starten und laufenlassen, er ist aber nicht so weit residentfähig, daß sein Code von mehreren Prozessen gleichzeitig ausgeführt werden darf.



## 4.2 Eingabe-Elemente

Die Maus ist das wichtigste Mittel zur Bedienung von DBug. Mit ihr können neue Fenster geöffnet oder Fensterdaten durch Anklicken manipuliert werden, wobei der Doppelklick eine besondere Rolle spielt. Die Tastatur ist nur für ergänzende Eingaben und Menü-Abkürzungen vorgesehen. Der Debugger verwendet ausgiebig Fenster zur Informationsausgabe. Zur Abfrage von Werten gibt es noch spezielle Fenster, sogenannte Requester.

### 4.2.1 Anklicken von Daten

Alle DBug-Fenster enthalten Texte mit Daten. Einige Textbereiche sind anklickbar und werden grafisch hervorgehoben, wenn Sie auf diese bei gedrückter, linker Maustaste zeigen. Angeklickte Werte dürfen Sie für weitere Operationen verwenden, indem Sie z.B. das Übernahme-Gadget eines anderen Fensters anklicken und damit dieses Fenster auf den angeklickten Wert einstellen (die Reaktion des Fensters hängt von seinem Typ ab). Drücken Sie während des Anklickens eine Alt-Taste, so multipliziert der Debugger den angeklickten Wert vor Gebrauch mit vier. Dabei wird der Wert aber nicht im Fenster oder Speicher geändert, sondern nur intern vor Benutzung mit vier multipliziert.

Sie können einen Wert nur so lange verwenden, wie er angeklickt ist. Anklicken einer anderen Stelle hebt die letzte Hervorhebung auf und den neu angeklickten Wert hervor. Sollte die neue Stelle nicht anklickbar sein, dann gibt es keinen angeklickten Wert.

Einige Funktionen des Debuggers verwenden einen angeklickten Wert, sofern ein solcher existiert, und bringen sonst einen Frage-Requester hervor, mit dem Sie zur Eingabe notwendiger Daten aufgefordert werden. Siehe auch unten folgenden Paragraphen 4.2.3.

Ein Doppelklick auf ein Datum bewirkt in der Regel bei allen Fenstern, daß ein Speicherfenster mit dem Datum als Startadresse geöffnet wird. Doppelklicken Sie also in einem Speicherfenster auf ein Langwort, das einen Zeiger enthält, so öffnet sich ein Fenster mit dessen Inhalt. Sollte es sich um einen BCPL-Zeiger handeln, müssen Sie nach dem oben Gesagten nur zusätzlich eine Alt-Taste drücken, um die korrekte Adresse zu erhalten. Soll das Fenster gleich mit dem Disassembler arbeiten, dann drücken Sie gleichzeitig die Control-Taste.

**Merkregel:** Einfacher Doppelklick öffnet Speicherfenster, mit gedrückter Shift-Taste können Sie editieren (siehe Status- und Speicherfenster), mit Alt-Taste wird ein BCPL-Zeiger referenziert und mit Control-Taste gleich der Disassembler eingeschaltet. Beachten Sie, daß <Alt> und <Control> auch beide gleichzeitig benutzt werden können und daß es von dieser Regel auch Ausnahmen gibt, z.B. das FPU-Statusfenster, bei dem ein Doppelklick auf das FPSR sofortiges Editieren erlaubt.

### 4.2.2 Fenster-Eigenschaften

Alle Fenster sind verschiebbar und lassen sich in den Vorder- bzw. Hintergrund legen. Die meisten Fenster lassen sich in ihrer Größe verändern und besitzen einen Satz von

Gadgets an der rechten Fensterumrandung. Zu diesen Gadgets gehört eine Kombination von einem Scrollbar mit Pfeil-Gadgets, mit der Sie Ausschnitte aus der Darstellung des jeweiligen Fensters anwählen. Die Art des Ausschnitts hängt vom Fenstertyp ab. Der Scrollbar zeigt durch seine Höhe immer die Ausschnittgröße bezogen auf die Gesamtgröße an Fensterdaten an, und seine Position indiziert die Ausschnittposition innerhalb der Datenmenge. Die Pfeil-Gadgets wählen die vorige bzw. nächste Darstellungszeile.

Wenn nicht editiert wird, dann simulieren bei den meisten Fenstern <Cursor-hoch> und <Cursor-runter> den Klick auf das entsprechende Pfeil-Gadget, d.h. zeilenweises Scrollen. Bei gleichzeitigem Drücken von <Shift> wird jeweils das Klicken über bzw. unter den Knopf des Scrollbars simuliert, d.h. seitenweises Blättern.

Das Übernahme-Gadget befindet sich zwischen Size-Gadget und dem Pfeil-Runter-Gadget. Wenn Sie es anklicken, wird entweder der gerade angeklickte Wert als Darstellungs-Startwert für das Fenster übernommen, oder (falls ein solcher nicht existiert) der Debugger fragt Sie über einen Requester nach dem Startwert.

Das Rückwärts-Gadget gibt es nur bei Speicherfenstern. Es stellt die Adresse ein Wort bzw. Byte rückwärts. Siehe Speicherfenster.

### 4.2.3 Requester

Requester sind spezielle Debugger-Fenster, die das Fortsetzen einer Operation erst nach Eingabe von Daten erlauben. Im Gegensatz zu Intuitions Requestern können Sie andere Teile des Debuggers bedienen, ohne einen Requester beantwortet zu haben, so können Sie z.B. ein Symbolfenster öffnen und einen Symbolwert nachgucken, um die neue Startadresse eines Speicherfensters anzugeben. Dabei ist es aber nicht möglich, weitere Requester aufzurufen. Es kann gleichzeitig nur ein einziger Requester geöffnet sein.

Sobald ein Requester erscheint, ist eines seiner Eingabefelder (dies sind Intuition-String-Gadgets) aktiviert und wartet auf Ihre Eingabe über die Tastatur, die übrigen sind leer oder aber bereits mit von Ihnen änderbaren Standardwerten initialisiert. Falls ein angeklickter Wert existiert, übernehmen einige Requester diesen in eines der Eingabefelder.

Sie können einen Requester ohne Eingabe beenden, indem Sie entweder sein Close-Gadget anklicken oder das Cancel-Gadget bedienen. Die Eingabe bestätigen Sie durch Drücken der Return-Taste oder durch Anklicken des Ok-Gadgets. Die gewählte Funktion wird erst ausgeführt, wenn alle Eingabewerte besetzt und auf Korrektheit überprüft sind.

Bei Falsch-Eingaben druckt der Debugger entsprechende Fehlermeldungen in die oberste Textzeile eines Requesters und beendet den Requester nicht, sondern verlangt eine Neueingabe. Dabei ist immer der Requester-Wert aktiviert, in dem der Fehler lokalisiert wurde. Dadurch können Sie einfach durch Drücken der Return-Taste vom letzten Eingabefeld zum nächsten leeren Eingabefeld springen, ohne die Maus zu benutzen. Wollen Sie allerdings einen vom Debugger vorgegebenen Wert ändern, müssen Sie mit der Maus arbeiten. In den Eingabefeldern gibt es die folgenden Editiermöglichkeiten:



**<Cursor-links>** bewegt den Cursor nach links.

**<Cursor-rechts>** bewegt den Cursor nach rechts.

**<Shift>+<Cursor-links>** setzt den Cursor auf den Stringanfang.

**<Shift>+<Cursor-rechts>** setzt den Cursor auf das Stringende.

**<Amiga>+<X>** löscht den String.

**<Amiga>+<Q>** restauriert den String so, wie er beim Editierbeginn war.

**<Delete>** löscht das Zeichen unter dem Cursor.

**<Backspace>** löscht das Zeichen links vom Cursor.

### 4.2.4 Integer-Ausdrücke

Sie dürfen bei Eingabe von Werten beliebige Ausdrücke angeben. Die Ausdruckauswertung ist weitgehend mit der des Optimierenden-Makro-Assemblers identisch (siehe auch OMA-Handbuch), es gelten aber alle Werte als absolute Integer, relative gibt es bei DBug nicht.

Ausdrücke werden hierarchisch ausgewertet, z.B. geht Punkt- vor Strichrechnung. Die Operanden können ein Symbolname sein (dieser wird automatisch durch seinen Wert ersetzt), der Name eines Daten- oder Adreßregisters (dieser wird durch den aktuellen Wert des Registers ersetzt) oder eine Zahl. Die Zahl kann in folgenden Zahlensystemen angegeben werden:

- dezimal ohne spezielle Angabe, z.B. »45«.
- hexadezimal mit einleitendem Dollarzeichen »$«, z.B. »$4e75«.
- oktal mit einleitendem At-Zeichen »@«, z.B. »@17«.
- binär mit einleitendem Prozentzeichen »%«, z.B. »%1001001011«.
- Stringkonstante in ASCII-Format. Diese beginnt mit einfachem oder doppeltem Hochkomma und muß auch mit diesem wieder enden, darf maximal vier Zeichen lang sein, wird rechtsseitig aufgefüllt und ermöglicht es, das einfassende Hochkomma in sich vorkommen zu lassen, wenn es doppelt angegeben wird. Z.B.: "th" 'lang' 'it"s'
- beim Assemblieren der aktuelle Programmzähler »*«.

Symbolnamen beginnen mit einem Buchstaben, Punkt oder einem Unterstrich. Nach dem ersten Zeichen sind außerdem noch Ziffern zugelassen. Lokale Label (siehe OMA-Handbuch) sind ebenfalls möglich, hinter dem Dollarzeichen »$« muß bei ihnen aber ihre einmalige Kennung angegeben werden (siehe solche Namen im Symbolfenster). Als Buchstaben sind auch die internationalen Zeichen des Amiga erlaubt, z.B. ist der Name »Zähler« korrekt.

Bitte beachten Sie, daß lokale Label in zu debuggenden Programmen nicht eindeutig sind, wenn diese aus mehreren Modulen bestehen. Dies kann speziell beim Assemblieren

in Speicherfenstern zu Meldungen wie »Out of range« o.ä. führen, da ein falsches, gleichnamiges Symbol herausgesucht wird.

Es wird immer mit ganzzahligen, vorzeichenbehafteten Werten von 32 Bit Länge gerechnet. Folgende Operatoren sind gültig:

| *Priorität* | *Operator* | *Bedeutung* |
|---|---|---|
| 5 | + | Monadisches Plus |
| 5 | - | Monadisches Minus |
| 5 | ~ | Logisches Negat (Tildezeichen) |
| 4 | >> | Rechtsverschieben (zweimal »>«) |
| 4 | << | Linksverschieben (zweimal »<«) |
| 3 | ^ | Logisches Exklusiv-Oder (»Dach«) |
| 3 | & | Logisches Und |
| 3 | \| | Logisches Oder |
| 3 | ! | Logisches Oder |
| 2 | % | Modulo |
| 2 | / | Division |
| 2 | * | Multiplikation |
| 1 | - | Subtraktion |
| 1 | + | Addition |

Operatoren mit höherer Priorität werden zuerst berechnet. Die Priorität kann durch Klammerung durchbrochen werden. Die Klammerverschachtelung reicht zusammen mit der Operatorverschachtelung bis zu 32 Ebenen tief. Die Operatoren mit der Priorität 5 sind monadische Operatoren, haben also nur einen Operanden, der unmittelbar auf den Operator folgt.

Nun noch ein paar Beispielausdrücke:

**3+4*5** ergibt 23

**'2'+start** ergibt $34, wenn "start" den Wert 2 hat

**"2'** ergibt einen Fehler, korrekt ist "2" oder '2'

***-start** ergibt 4, wenn "start" = 2 und der Programmzähler = 6 ist

**~1** ergibt $FFFFFFFE = -2

**(8>>2)** acht, zweimal nach rechts geshiftet (ergibt 2)

**(1<<7)** eins, siebenmal nach links geshiftet (ergibt $80)

**(1<<7)!1** ergibt $81

**(80+128)<<16!(32+128)<<4** ergibt $00D00A00

### 4.2.5 Fließkomma-Ausdrücke

Beim Assemblieren von FPU-Befehlen und Eingaben im FPU-Statusfenster können auch Fließkomma-Ausdrücke ausgewertet werden, und es sind dann Fließkommazahlen



zugelassen. Als Operatoren sind dann nur noch die Grundrechenarten »+«, »-«, »*«, »/« und der Modulo-Operator »%« mit der gleichen Priorität wie oben angegeben zugelassen. Dabei sind auch Symbolnamen oder Daten- oder Adreßregisternamen erlaubt, die in Fließkommawerte umgewandelt werden.

Die Fließkommazahlen werden im üblichen Format spezifiziert, d.h. zuerst ein Vorzeichen, dann eine Zahl mit oder ohne Kommastelle (als Punkt anzugeben), dann ein »E« oder »e« zur Einleitung des Exponenten und daraufhin der Exponent selber, evtl. ebenfalls mit Vorzeichen. Der Wertebereich geht bei ca. 19 Stellen Genauigkeit von $6*10^{4931}$ bis $9*10^{-4952}$. Wie bei Integer-Ausdrücken wird ein Fließkomma-Ausdruck hierarchisch berechnet und kann ebenfalls bis zu 32 Ebenen tief verschachtelt werden.

Beispiele für Fließkommazahlen und -ausdrücke:

- 3.1415926535*24.3+8
- -5.61e-25
- .00004592834
- +3

Beachten Sie, daß Sie Fließkomma-Ausdrücke nur verwenden können, wenn Sie einen Fließkomma-Coprozessor vom Typ MC68881 oder MC68882 oder eine MC68040-MPU in Ihrem Rechner eingebaut haben.

## 4.3 Fenstertypen

DBug besitzt mehrere Typen von Fenstern. Es gibt u.a. Speicherfenster und registeranzeigende Statusfenster. Im folgenden werden die Eigenheiten der verschiedenen Typen beschrieben.

### 4.3.1 Das Speicherfenster

Dieses Fenster ist das wichtigste von allen. Mit ihm haben Sie die Möglichkeit, Daten verschieden formatiert im Speicher anzusehen. Eine Zeile beginnt in der Regel mit der Speicheradresse, darauf folgt der Datenteil und optional ein Textblock, der die Daten in ASCII-Form sichtbar macht. Die Daten werden zehnmal pro Sekunde auf den neuesten Stand gebracht. Dadurch können sich ständig ändernde Wert beobachtet werden, oder Sie können zugucken, wie Ihr Programm arbeitet.

Das Adressenformat ist einstellbar (siehe Address-Menü) und hat regulär sechs Stellen bei einem MC68000/010-Prozessor und acht Stellen bei den höheren Prozessoren. Falls es ein Symbol mit der Speicheradresse gibt, wird eine Symbolzeile mit dessen Namen vor der Datenzeile eingeschoben. Sofern ein Symbol mit seiner Adresse zwischen dem Beginn eines Datums und seinem Ende liegt, erhält es ebenfalls eine eigene Zeile, ist dann aber eingeklammert.

Der Datenteil enthält die Daten in wählbarem Format (siehe Data-Menü), dabei existiert auch das Einschalten eines Disassemblers für die Motorola-Prozessoren. Der Disassembler unterstützt alle MPUs einschließlich Fließkomma-Coprozessoren. Als PMMU-Befehle existieren nur die der MC68030, da das einzige Amiga-Board, das die MC68851 PMMU benutzt (die noch viel mehr Befehle und Register als die MC68030 hat), das Commodore-Board A2620 ist. Falls möglich, werden im Disassembler-Listing Adressen durch Symbolnamen ersetzt, so daß der Code wieder leicht lesbar wird. Darüberhinaus erscheinen Quell- und Zieladresse als Kommentare hinter den Instruktionen. Die Werte des Disassemblers sind alle anklickbar, doppelklicken Sie z.B. einen Variablennamen, dann öffnet sich ein Speicherfenster mit der Adresse der Variablen als Startadresse. Beim Disassembler ist die Adresse der nächsten auszuführenden Instruktion zusätzlich mit einem Stern gekennzeichnet. Handelt es sich sogar um ein PC-Disassembler-Fenster, dann folgt dessen Inhalt immer dem aktuellen PC.

Das Rückwärts-Gadget über dem Scrollbar bewirkt, daß die Startadresse des Fensters um ein Wort zurückgeschaltet wird. Ist der Disassemblermodus abgeschaltet und Bytegröße eingeschaltet, dann wird ein Byte zurückgegangen.

Durch Bedienung des Übernahme-Gadgets wird der gerade angeklickte Wert als Startadresse übernommen. War nichts angeklickt, geht der Frage-Requester auf. Ist es Hunk-Range, dann wird der Bereich sofort auf den Hunk gesetzt, in dem die Adresse liegt. Bei Absolute-Range wird gemeckert (wenn Frage-Requester) bzw. gebeept, wenn die Adresse nicht im gewählten Bereich ist.

Ein normaler Doppelklick öffnet ein Speicherfenster mit der angeklickten Adresse als Startadresse. Ist es ein Opcode beim Disassembler, so springt ein Cursor in die Zeile, und Sie haben die Möglichkeit, Instruktionen zu editieren und zu assemblieren (siehe



Editiermöglichkeiten unten). Der Assemblerteil besitzt dieselben Fähigkeiten wie der Disassembler und stellt, falls nötig, den Cursor auf die Stelle, an der Sie evtl. einen Fehler gemacht haben. Die Fehlermeldung finden Sie dann in dem Fenstertitel.

Ist es bei ausgeschaltetem Disassembler ein Doppelklick auf ein Datum mit gedrückter Shift-Taste, dann geht ein Cursor in die Zeile und erlaubt Editieren der Daten wie in einem Intuition-String-Gadget einschließlich Cursorpositionierung mit der Maus wie folgt:

**<Cursor-links>** bewegt den Cursor nach links.

**<Cursor-rechts>** bewegt den Cursor nach rechts.

**<Shift>+<Cursor-links>** setzt den Cursor auf den Stringanfang.

**<Shift>+<Cursor-rechts>** setzt den Cursor auf das Stringende.

**<Amiga>+<X>** löscht den String.

**<Amiga>+<Q>** restauriert den String so, wie er beim Editierbeginn war.

**<Delete>** löscht das Zeichen unter dem Cursor.

**<Backspace>** löscht das Zeichen links vom Cursor.

Die Daten werden bei Verlassen der Zeile in den Speicher geschrieben. Sie brechen das Editieren durch Drücken der Escape-Taste oder Wählen einer anderen Funktion (z.B. eines Menüs) ab. Die Return-Taste schreibt die Daten und beendet das Editieren. Mit <Cursor-hoch> und <Cursor-runter> wählen Sie zusätzlich die vorige bzw. nächste Zeile an, sofern der Bereich nicht überschritten wird. Bevor die Daten geschrieben werden, werden sie u.a. auf korrekten Bereich geprüft. Ist etwas falsch, dann wird eine Fehlermeldung in den Fenstertitel gedruckt und die Zeile nicht verlassen.

Es ist noch wichtig zu wissen, daß der durch Speicherfenster und BigStatus-Fenster erreichbare Adreßraum eingeschränkt ist. Wenn nämlich ein Zugriff durch den Debugger außerhalb des gültigen Adreßraums erfolgt, wird durch dauernde Exception-Auslösung z.B. auf einem Amiga 3000 der ganze Rechner extrem stark verlangsamt. Daher wird intern automatisch nur der adressierbare, vorhandene Speicher sowie der Bereich $F80000 bis $FFFFFF zugelassen. Weitere Bereiche müssen in Textfiles abgelegt werden, die DBug.memory heißen und in dem Directory sein dürfen, aus dem der Debugger gestartet wird, sowie in dem DEVS:-Directory. Falls sowohl im aktuellen Directory als auch im DEVS:- Directory ein Speicherfile vorhanden ist, werden beide eingelesen und deren Bereiche überlagert. Dadurch kann man einmal global einen Speicherbereich anwählbar machen (DEVS:) und einmal lokal (aktuelles Directory). Eine Zeile dieser Textfiles hat das Format:

```
$E00000 - $E00FFF  ;dies ist ein Kommentar
...
(weitere Zeilen)
```

Als Trennzeichen sind Leer- oder Tabualtorzeichen zugelassen. Ein Zeilenende wird durch Linefeed oder Carriage Return erkannt. Nullbytes werden durch Spaces ersetzt. Das File darf maximal 4095 Bytes lang sein. Sollte das File irgendwie falsch sein, beept das Display. Die Endadresse gilt als einschließlich.

### 4.3.2 Das Statusfenster

Das Statusfenster gibt Ihnen den wichtigen Einblick in den Registerzustand des Prozessors. Es stellt alle Daten- und Adreßregister, den Programmzähler PC, die Statusflags sowie den als nächstes auszuführenden Befehl disassembliert dar. Darüberhinaus bekommen Sie hier Statusinformation des Debuggers in der untersten Zeile präsentiert.

Die obersten vier Zeilen enthalten automatisch den aktuellen Zustand der Daten- und Adreßregister. Sie ändern einen Registerwert, indem Sie ihn bei gedrückter Shift-Taste doppelklicken. Ein Requester erscheint, mit dem Sie den neuen Wert als beliebigen Ausdruck eingeben können. Ferner laden Sie ein Register sofort mit dem zuletzt angeklickten Wert, indem Sie seinen *Namen* doppelklicken. Dies funktioniert mehrfach, so daß Sie den Wert hintereinander in mehrere Register schieben können.

Die fünfte Zeile enthält den Programmzähler PC des nächsten Befehls, der ausgeführt wird, sowie die fünf Flags des Prozessor-Condition-Code-Registers. Den PC können Sie wie die Register oben ändern, z.B. klicken Sie in einem Disassembler-Fenster die Adresse einmal an, bei der die Programmausführung fortgesetzt werden soll, und danach den Namen »PC«, so daß der PC auf diese Adresse gesetzt wird. In einem Disassembler-Fenster wird der aktuelle PC durch einen Stern vor dem Opcode gekennzeichnet. Die Flags ändern Sie durch einen Doppelklick; dabei wird der Zustand des Flags invertiert.

In der sechsten Zeile wird der Befehl disassembliert, der als nächstes ausgeführt wird. Als Kommentar werden die Adressen ausgeben, die sich durch den momentanen Registerzustand berechnen, sowie bei Verzweigungen und anderen, zustandsabhängigen Befehlen TRUE bzw. FALSE abhängig davon, ob die Befehlsbedingung beim momentanen Flagzustand zutrifft. Wenn Sie den Opcode des Befehls doppelklicken, können Sie einen anderen Befehl an die PC-Adresse assemblieren (siehe Speicherfenster: Disassembler/Assembler). Beachten Sie jedoch, daß ein neuer Befehl eine andere Länge haben und Folgebefehle teilweise überschreiben kann.

Die siebte Zeile enthält schließlich eine Meldung über den aktuellen Status des Debuggers (z.B. ob gerade ein Programm geladen wird oder eine Exception vorkam).

Bei allen Werten bewirkt ein normaler Doppelklick das Öffnen eines Speicherfensters mit dem angeklickten Wert als Startadresse.

### 4.3.3 Das BigStatus-Fenster

Dieses Fenster ist im wesentlichen mit dem Statusfenster identisch, hat aber als Erweiterung einen Speicherauszug des Bereiches, auf den die Adreßregister zeigen. Dieser Auszug umfaßt drei Langworte sowie einen Textblock. Während Sie ein Programm einzelschrittweise ausführen lassen, können Sie somit beobachten, an welchen Stellen die Adreßregister arbeiten und was sich dort befindet.

Im Gegensatz zu Speicherfenstern werden diese Daten hier nicht automatisch refresht, eine Änderung der angezeigten Speicherzellen wird erst sichtbar, wenn das Statusfenster selber refresht wird (z.B. nach einem Single-Step).



### 4.3.4 Das FPU-Statusfenster

Das FPU-Statusfenster können Sie nur benutzen, wenn Sie einen Fließkomma-Coprozessor vom Typ MC68881 oder MC68882 oder eine MC68040-MPU in Ihrem Rechner eingebaut haben. Sollte dies nicht der Fall sein, meldet Ihnen das FPU-Fenster dies und verbirgt seine Funktionen vor Ihnen.

Das Fenster stellt den Inhalt der FPU-Register FP0 bis FP7 sowie den Status des Coprozessors dar und ermöglicht Ihnen somit, Fließkommaberechnungen zu verfolgen. Normale Werte werden normalisiert in Exponentialdarstellung ausgegeben, wobei es 18 Nachkommastellen und einen vierstelligen Exponenten zur Basis zehn gibt. Darüberhinaus gibt es die Werte »inf« für die unechte Zahl »Unendlich« und »NAN« für »Not-A-Number« (siehe auch FPU-Handbuch).

Sie ändern die Fließkommawerte, indem Sie einen Wert doppelklicken, woraufhin ein Requester erscheint, in dem Sie den neuen Wert editieren können. Dabei dürfen Sie einen beliebigen Fließkomma-Ausdruck benutzen.

Das Statusregister FPSR können Sie - ebenfalls durch Doppelklick - wie eine normale Langwortzahl editieren. Vier Flags dieses Registers werden zudem gesondert dargestellt. Diese können Sie durch Doppelklick in ihrem Zustand ändern, dabei paßt sich der Wert des FPSR automatisch an.

### 4.3.5 Das Hunk-Fenster

Sofern gerade ein Programm debuggt wird, zeigt das Hunk-Fenster die Segmente des Programmes an. Die Anzeige umfaßt den Segmentbeginn, d.h. die Startadresse, die Segmentgröße, dann den Hunknamen (wenn dieser im Programmfile über hunk_name oder hunk_unit angegeben ist; meistens nicht vorhanden) und daraufhin eingeklammert einen Symbolnamen, der auf der Startadresse liegt. Sowohl der Hunk- als auch der Symbolname können fehlen.

Bei Bedienung des Übernahme-Gadgets wird der Hunk als erster angezeigt, in dem die eingegebene Adresse liegt. Gibt es keinen solchen Hunk, dann beept das Display einmal. Beim Frage-Requester erfolgt eine Fehlermeldung, und der Requester wird nicht verlassen.

Durch einen Doppelklick wird ein Speicherfenster mit der entsprechenden Startadresse geöffnet. Wurde der Hunkname doppelgeklickt, dann hat das Speicherfenster diesen als Fensternamen.

Bitte beachten Sie, daß es für Tasks, die keine Prozesse sind, keine Segmente gibt und das Hunk-Fenster daher leer bleibt.

### 4.3.6 Das Symbol-Fenster

In diesem Fenster werden die Symbole des gerade zu debuggenden Programmes dargestellt. Auf die Symboladresse folgt jeweils der Symbolname. Das Zusatzmenü **Order**

gibt Ihnen die Möglichkeit, die Symbole nach aufsteigenden Adressen oder alphabetisch zu sortieren. Dabei hat jedes Symbolfenster seine eigene Sortierung.

Bei Bedienung des Übernahme-Gadgets wird das Symbol mit der entsprechenden Adresse als erstes angezeigt. Gibt es kein solches Symbol, dann beept das Display einmal. Beim Frage-Requester erfolgt eine Fehlermeldung, und der Requester wird nicht verlassen.

Durch einen Doppelklick wird ein Speicherfenster mit der entsprechenden Startadresse geöffnet. Bitte beachten Sie, daß das Symbolfenster leer ist, wenn keine Symbole für das Programm gefunden wurden, und daß an die lokalen Label des Assemblers eine spezielle Kennung zur eindeutigen Identifizierung angehängt ist.

### 4.3.7 Das Breakpoint-Fenster

Die von Ihnen gesetzten Breakpoints (siehe auch Break-Menü) können Sie hier betrachten. Auf die Breakpoint-Adresse folgt das Enable/Disable-Flag (»+« für enabled, »-« für disabled), dann ein Zähler, der von Null (oder einem von Ihnen angegebenen Wert) hochgezählt wird, bis der Anhaltewert (nach dem »/«) erreicht ist. Tritt ein Breakpoint auf, wird der Zähler erhöht und das Programm nur dann unterbrochen, wenn der Zähler den Anhaltewert erreicht hat! Falls es ein Symbol für die Breakpoint-Adresse gibt, wird dieses auch ausgegeben.

Bei Bedienung des Übernahme-Gadgets wird der Breakpoint mit der entsprechenden Adresse als erster angezeigt. Gibt es kein solches Symbol, dann beept das Display einmal. Beim Frage-Requester erfolgt eine Fehlermeldung, und der Requester wird nicht verlassen.

Durch Doppelklick können Sie den Zähler und den Anhaltewert ändern, dabei darf der Anhaltewert aber nicht Null sein, und der Zähler muß immer kleiner gleich dem Anhaltewert sein. Doppelklicken auf das Enable/Disable-Flag kehrt dessen Zustand um. Durch einen Doppelklick auf die Adresse bzw. den Symbolnamen wird ein Speicherfenster mit der entsprechenden Startadresse geöffnet.

Bei Speicher-Breakpoints gibt es neben den oben erwähnten Ausgaben noch die Größenangabe (»B«, »W« oder »L«), die Relation (»==«, »!=«, »<=«, »>=«) und die betroffene Speicheradresse. Alle drei sind nicht anklickbar.

### 4.3.8 Das History-Fenster

Die History dient zum Aufzeichnen von Befehlen während der Abarbeitung des zu debuggenden Programmes. Wenn Sie nach Fehlern suchen, geraten Sie oft über den Punkt hinaus, an dem das falsche Programmverhalten das erste mal auftritt. Damit Sie das Debuggen nicht von vorne beginnen müssen, können Sie über die History verfolgen, welche Befehle das Programm ausgeführt hat und welche Registerinhalte zu den jeweiligen Zeitpunkten vorlagen. Die Aufzeichnungstiefe ist einstellbar und die Aufzeichnung ein- bzw. ausschaltbar (siehe auch Trace-Menü). Sobald die Aufzeichnungstiefe erreicht



ist, werden die ersten aufgezeichneten Elemente wieder überschrieben. Sie haben damit zwar nur eine Aufzeichnung der letzten x Befehle, die Tiefe ist dafür nur durch den verfügbaren Speicher beschränkt. Nach dem Start ist eine Tiefe von zehn eingestellt, damit auch Besitzer von Rechnern mit nur 512 KByte die Möglichkeit haben, den Debugger zu benutzen.

Ein History-Element umfaßt einen kompletten Registersatz der MPU (die FPU wird aus Platzgründen nicht aufgezeichnet) mit einem PC und der dort stehenden, disassemblierten Prozessorinstruktion. Damit Sie Änderungen in den Registern leicht verfolgen können, besitzt das History-Fenster kein weiches Scrollen für die Pfeil-Gadgets, sondern schaltet jeweils ein komplettes Element weiter.

Das Übernahme-Gadget übernimmt hier die Nummer eines anzuzeigenden Elementes. Sollte die Aufzeichnungstiefe geringer als die angegebene Nummer sein, beept das Display einmal. Beim Frage-Requester erfolgt eine Fehlermeldung, und der Requester wird nicht verlassen.

Auch hier bewirkt ein Doppelklick das Öffnen eines Speicherfensters mit dem angeklickten Wert als Startadresse.

### 4.3.9 Das Text-Fenster

Das Text-Fenster wurde geschaffen, damit Sie auch an Information über interne Daten des Betriebssystems kommen können. Befehle werden wie in einem CLI-Fenster eingegeben, wobei das Editieren wie oben angegeben dem der Intuition-String-Gadgets gleicht. Groß- und Kleinschreibung der Befehle ist unwichtig, es sind die folgenden verfügbar:

- **?** Bietet eine Aufstellung der verfügbaren Befehle.
- **d** Gibt Liste der Devices aus.
- **i** Gibt Information über die zu debuggende Task aus. Hängen Sie eine Adresse in hexadezimaler Form mit »i $Adresse« an, so erhalten Sie die Information über die korrespondierende Task mit dieser Adresse. Diese Information umfaßt die Daten des Task-Kontrollblocks, der Process- und der CLI-Struktur. Bitte beachten Sie, daß in der Information alle BCPL-Zeiger schon auf Prozessoradressen umgerechnet worden sind.
- **l** Gibt Liste der Libraries aus.
- **p** Gibt Liste der Ports aus.
- **r** Gibt Liste der Resources aus.
- **s** Gibt Liste der Interruptserver aus.
- **t** Gibt Liste der Tasks aus.

Die letzten 64 Zeilen des Fensters werden gepuffert und können über Scrollbar und Pfeil-Gadgets wieder sichtbar gemacht werden. Um dann neue Befehle einzugeben, brauchen Sie nicht zum Pufferende zu gehen, neue Ausgaben schalten automatisch an das Pufferende und erzeugen dort die neuen Zeilen. Sollte der Cursor zum Eingeben nicht sichtbar sein, müssen Sie nur die unterste Textzeile des Fensters anklicken.

Das Übernahme-Gadget hat hier keine Funktion. Durch einen Doppelklick auf einen Wert wird ein Speicherfenster mit der entsprechenden Startadresse geöffnet.

### 4.3.10 Das Source-Level-Fenster

DBug bietet auch echtes Source-Level-Debugging. Wenn Sie ein Programm in den Debugger laden, das unter Verwendung der Startoption »-c q« des Optimierenden-Makro-Assemblers erzeugt wurde, dann erscheint die Sourve-Level-Information, nämlich der Quelltext des assemblierten Programmes, in den Source-Fenstern.

Beim Öffnen des Source-Fensters kann eine Adresse angeklickt sein, dann wird automatisch auf die passende Sourcestelle positioniert, sofern es eine für diese Adresse gibt. Das Positionieren kann auch über das Übernahmegadget geschehen. Die Adressen vor den Quellzeilen sind anklickbar und können zum Speicherfenster-Öffnen bzw. Breakpoint-Setzen benutzt werden. Wenn der aktuelle PC die Adresse einer Quellzeile enthält, wird diese per Invertieren markiert. Normalerweise wird das Source-Fenster immer mit dem aktuellen PC mitgeführt, das kann aber mit Don't move im Source-Menü an- bzw. abgeschaltet werden. Die Darstellung der Quellzeilen kann Tabulatorzeichen enthalten; über Tabulator im Source-Menü kann eingestellt werden, auf welche Tabulatorpositionen diese Zeichen expandiert werden (Default ist 8). Schließlich können noch Strings im Quelltext mittels Find string beginnend ab der zweiten sichtbaren Zeile gefunden werden. Der Vergleich findet ohne Groß-/Kleinschreibung statt.

Das Übernahme-Gadget bewirkt, daß die entsprechende Source-Level-Zeile zur Adresse gesucht wird. Falls für bestimmte Programmstellen keine Source-Level-Information vorliegt, kann natürlich auch keine im Source-Fenster dargestellt werden.



## 4.4 Debuggen von Tasks

Mit DBug können Sie beliebige Tasks unter die Kontrolle des Debuggers bringen. DBug ermöglicht Ihnen sogar, vom Hauptprogramm abgetrennt laufende Tasks zu debuggen, die Sie normalerweise nicht erreichen würden. Wenn Sie den Debugger mehrfach gleichzeitig gestartet haben, können Sie sogar Haupt- und Nebentask gleichzeitig debuggen. In diesem Kapitel erfahren Sie, auf welche Weise verschiedene Tasks oder Programme abgefangen und vom Debugger kontrolliert werden.

### 4.4.1 Abfangen der nächstgestarteten Task

Wollen Sie eine Task abfangen, die von der Workbench oder dem CLI oder einer Shell aus gestartet wird, so wählen Sie den Menüpunkt Catch WB/CLI im Project-Menü. Falls Sie ein Programm unter einem CLI oder einer Shell unter Kickstart 1.3 benutzen, müssen Sie den Programmnamen so weit angeben, daß der Debugger das Programmfile von dem Directory aus, von dem er gestartet wurde, finden kann. Ab System 2.0 ist dies nicht mehr notwendig.

Beachten Sie, daß Sie ein zu debuggendes Programm unter dem CLI oder einer Shell nicht mit dem *Run*-Kommando starten, da sonst das *Run*-Kommando abgefangen wird. Ferner können Sie beim Start des Debuggers vom CLI aus gleich ein Programm zum Debuggen anwählen, indem Sie die Startzeile des Programms einfach an den Debuggernamen anhängen.

Wollen Sie hingegen die nächste mit der Betriebssystemfunktion *AddTask()* oder *CreateProc()* gestartete Task abfangen, dann wählen Sie den Menüpunkt Catch next task im Project-Menü. Dabei sollten Sie aber genau wissen, welche Tasks demnächst entstehen, sonst erwischen Sie unter Umständen die falsche.

### 4.4.2 Abfangen einer bereits laufenden Task

Gehen Sie in ein Text-Fenster und lassen Sie sich die Taskliste über das »t«-Kommando ausgeben. Klicken Sie die Adresse der zu debuggenden Task einmal an und wählen Sie den Menüpunkt Select task im Project-Menü. Die Task wird sofort angehalten; wartet sie jedoch auf Signale (z.B. auf einen Gadget-Druck), müssen Sie diese Signale von Hand auslösen, damit das Abfangen stattfinden kann. Sofern Sie die Taskadresse auswendig oder schon aus anderen Quellen wissen, können Sie auch gleich Select task wählen und die Taskadresse über den dann erscheinenden Requester eingeben. *Wählen Sie keine Systemtasks an, da der Rechner sonst abstürzen bzw. verklemmen kann.*

### 4.4.3 Abfangen einer abgestürzten Task

Leider ist diese Fähigkeit nur mit Kickstart 1.3 oder kleiner vorhanden, da ab System 2.0 umfangreiche Änderungen vorgenommen wurden, so daß keine nachträgliche Abfangmöglichkeit mehr besteht.

*Machen Sie dies nur, wenn eine Task abgestürzt ist und Sie diese sofort auswählen können.* Tasks, die ohne Debugger-Kontrolle eine Exception auslösen, werden von Exec angehalten, und der DOS-Requester »Software error...« erscheint im Screen der Workbench.

Sie starten dann den Debugger (oder hatten ihn im Hintergrund) und gehen wie im vorigen Abschnitt vor, wählen aber den Menüpunkt Postmortem statt Select task. Nun müssen Sie nur noch *Cancel* im DOS-Requester dieser Task anklicken, und der Debugger zeigt Ihnen die Exception und den fehlerhaften Befehl an. *Sie sollten unbedingt nach* Postmortem *das Cancel-Gadget klicken, ferner ist es empfehlenswert, nach erfolgter Absturzursachen-Suche den Rechner neu zu booten.* Es ist allerdings gut möglich, daß Sie den Betrieb über Kill task fortsetzen oder das Programm nach Reparatur normal beenden können.

### 4.4.4 Unterbrechen des Abfangens

Mit dem Befehl Cancel catching kann jegliches Abfangen abgebrochen werden.

### 4.4.5 Nach dem Abfangen

Sobald DBug eine Task abgefangen hat, versucht er, das Programmfile der Task zu finden. Ist dies möglich, dann lädt er die Symbole des Programmes (diese haben Sie z.B. mit dem Optimierenden-Makro-Assembler mit der »-c s«-Startoption erzeugt), evtl. die Source-Level-Information (diese bekommen Sie durch OMAs Startoption »-c q«) und zeigt den als nächstes auszuführenden Befehl in den Statusfenstern an. Sie können nun die Variablen und Funktionen der Task mit ihrem symbolischen Namen ansprechen (der Disassembler ersetzt dann z.B. Speicheradressen durch symbolische Namen) oder gleich die komfortable Source-Level-Ebene benutzen. Falls das Programmfile nicht gefunden wird oder dieses keine Symboldefinitionen enthält, meldet der Debugger dies. Beachten Sie, daß es möglich ist, mehrere gleichnamige Symbole zu haben, so daß Sie mit dem symbolischen Namen unter Umständen die falsche Adresse ansprechen, und daß reine Tasks, die kein Prozeß sind, kein Programmfile und daher auch keine Symbole besitzen.

Sie haben außerdem die volle Kontrolle über die Ausführung der Task. Sehen Sie sich das Trace-Menü an. Mit Single step führen Sie den gerade in einem Statusfenster sichtbaren Befehl aus und halten das Programm sofort wieder an. Welcher Befehl dies ist, können Sie u.a. in einem Disassembler-Fenster erfahren, denn dieser ist dort mit einem Stern gekennzeichnet. Trace calls arbeitet ähnlich, führt aber durch BSR oder JSR aufgerufene Unterprogramme mit voller Geschwindigkeit aus und hält die Task erst wieder nach der Rückkehr aus dem Unterprogramm an. Go läßt die Task normal loslaufen und unterbricht sie nur, wenn eine Exception oder ein Breakpoint vorkam.

Falls Sie ein PC-Disassembler-Fenster geöffnet haben, folgt dieses während der Einzelschritte dem aktuellen PC, so daß die gerade zu bearbeitende Stelle immer deutlich sichtbar ist. In einem Source-Level-Fenster können Sie sogar gleichzeitig den Quelltext ihres Programmes sehen. Auch dort ist der aktuelle PC markiert.



Solange eine Task unter Kontrolle des Debuggers läuft, kann sie nicht abstürzen, denn es werden die geläufigsten Ausnahmezustände (Exceptions) abgefangen. Der Debugger unterbricht einfach das Programm und sagt Ihnen, welche Exception das Anhalten an welcher Stelle verursacht hat. Es gibt auch Exceptions (speziell vom Betriebssystem verursachte), die sofort zu einem Guru führen. Die folgenden Exceptions werden von DBug verarbeitet:

- Address error
- Illegal instruction
- Zero divide
- CHK, CHK2
- TRAPV, TRAPcc, cpTRAPcc
- Privilege Violation
- Line-A- und Line-F-Emulator
- alle FPU-Exceptions

### 4.4.6 Beenden der Task

Eine Task beendet sich selber durch ein RTS bei passendem Stackpointer oder durch die DOS-Funktion *Exit()*. Der Debugger druckt eine entsprechende Meldung und blockiert weiteres Debuggen, so daß Sie dann eine neue Task anwählen können. Sollte die Task an einen Punkt kommen, an dem Sie sie nicht normal beenden können, steht Ihnen noch der Befehl Kill task zur Verfügung, der die Ausführung der Task abbricht und ihren Code aus dem Speicher entfernt. Von der Task beanspruchte Ressourcen oder z.B. Intuition-Windows bleiben dabei allerdings belegt. Dafür können Sie sich eine Fluchttür offenhalten, um beispielsweise noch in ein CLI oder eine Shell zu gelangen und Dateien auf Platte oder Diskette zu retten.

## 4.5 Die Menüs

### 4.5.1 Das Project-Menü

Catch WB/CLI

Der Debugger fängt die nächste vom CLI, der Shell oder der Workbench gestartete Task ab. Dabei dürfen die CLI-Tasks nicht mit *Run* gestartet werden, da sonst das Kommando *Run* debuggt wird. Beachten Sie, daß Sie unter Kickstart 1.3 den Programmnamen so weit angeben, daß der Debugger das Programmfile finden kann. Siehe auch Kapitel 4.4.

Catch next task

Der Debugger fängt die nächste von der Exec-Funktion *AddTask()* oder *CreateProc()* gestartete Task ab. Sie sollten hierbei wissen, welche Tasks demnächst entstehen.

Cancel catching

Dies bricht das obige Abfangen ab. Sie können sofort wieder Programme ohne Kontrolle durch den Debugger starten.

Select task

Diese Funktion ermöglicht Ihnen, eine bereits laufende Task zum Debuggen auszuwählen. Dazu klicken Sie die Taskadresse einfach im Textfenster an und wählen diesen Menüpunkt. Die Task wird sofort angehalten, und es wird nach ihren Symbolen gesucht. Siehe auch Kapitel 4.4.

Postmortem

Diese Funktion arbeitet ähnlich wie Select task, wählt aber eine Task zum Debuggen, die bereits mit einem Software-Fehler von Exec angehalten wurde. Wählen Sie die Taskadresse und diesen Menüpunkt an und klicken Sie auf *Cancel*. Der Debugger übernimmt die Kontrolle (siehe auch »Debuggen von Tasks«). Leider ist dies nur mit Kickstart 1.3 oder kleiner möglich, nicht ab System 2.0. *Sie sollten unbedingt nur bereits verstorbene Tasks über diesen Menüpunkt anhalten und die Anwahl nach Auswahl der Adresse unverzüglich durchführen.*

Leave task

Gibt die Kontrolle über die aktuelle Task ab und läßt diese frei loslaufen. Sollte nun noch ein Fehler auftreten, kann dieser höchstens noch über Postmortem oder Select task in Erfahrung gebracht werden.

Kill task

Die abgefangene Task wird einfach beendet und entfernt. Von der Task belegte System-Ressourcen u.ä. werden nicht freigegeben, so bleiben z.B. Intuition-Windows dieser Task einfach »in der Luft hängen« und geöffnet. Bitte beachten Sie, daß Kill task nicht den perfekten Ausstieg aus dem laufenden Programm bietet. Speziell workbench-gestartete Tasks können beim Aufruf dieses Kommandos Probleme bereiten, da ihre Workbench-Message nicht beantwortet werden kann.



Name log file

DBug besitzt ein sogenanntes Logfile, in das Ausgaben geschrieben werden können. Mit dieser Funktion geben Sie dem Logfile einen Namen. Sobald ein Name bekannt ist, kann das Logfile geöffnet werden. Solange es nicht offen ist, kann der Name abgeändert werden. Sie können das File auch schließen und ein zweites File mit anderem Namen eröffnen. Der Name darf maximal 127 Zeichen lang sein.

Open log file

Öffnet das Logfile mit dem angegebenen Namen. Sollte dies nicht funktionieren, beept das Display und informiert Sie so. Beachten Sie, daß die Ausgaben in das Logfile an ein evtl. altes, bereits existierendes File angehängt werden. Ein altes File gleichen Namens wird also nicht gelöscht!

Close log file

Schließt das Logfile, sofern es offen ist.

Log quiet, Log verbose

Diese Punkte bewirken das An- bzw. Abschalten von Meldungen während des Tracens und Steppens. Die Debugging-Aktionen können damit in dem Logfile protokolliert und später in Ruhe analysiert werden.

Quit

Beendet den Debugger. Wird gerade eine Task debuggt, dann wird vorher noch Kill task ausgeführt. Ist das Logfile noch offen, wird es geschlossen.

### 4.5.2 Das Open-Menü

Dieses Menü öffnet Fenster mit dem angegebenen Typ. Haben Sie einen Wert angeklickt, dann wird er automatisch als Startwert für das Fenster genommen, z.B. können Sie beim Öffnen eines Symbolfensters gleich feststellen, ob es ein Symbol mit der entsprechenden Adresse gibt.

### 4.5.3 Das Trace-Menü

Single step

Dieser Befehl führt einen Einzelschritt in der zu debuggenden Task aus. Sollte es die letzte Programminstruktion vor dem Terminieren sein, dann wird die Task beendet und eine entsprechende Meldung ausgegeben. Diese Funktion kann auch erreicht werden, indem Sie die Funktionstaste <F1> drücken.

Wenn das Logfile geöffnet und Log verbose gewählt ist, dann wird eine Protokollzeile in das Logfile gegeben.

Trace calls

Dies funktioniert ähnlich wie Single step, arbeitet aber durch BSR oder JSR verursachte Unterprogrammaufrufe mit normaler Geschwindigkeit ab und kehrt danach zurück. Bitte beachten Sie, daß dies auch für Unterprogrammaufrufe im ROM gilt! Die Funktionstaste <F2> wählt diese Funktion ebenfalls an.

Sollte ein Breakpoint oder eine andere Unterbrechung im Unterprogramm auftreten, dann hält der Debugger automatisch bei der späteren Rückkehr aus dem Unterprogramm an. D.h. Sie müssen nicht den Rest der Unterprogrammes nach der Unterbrechung einzelschrittweise durchsteppen, sondern können durch Go schnell und sicher zurückgelangen.

Wenn das Logfile geöffnet und Log verbose gewählt ist, dann wird eine Protokollzeile in das Logfile gegeben.

Skip code

Überspringt den aktuellen Befehl ohne Ausführung. Die Funktionstaste <F3> wählt diese Funktion ebenfalls an.

Go

Arbeitet die Task ab dem aktuellen PC mit normaler Geschwindigkeit ab. Falls Sie keine Breakpoints gesetzt haben, fragt der Debugger vorher über einen Requester, ob Sie tatsächlich loslaufen möchten. Für Go gibt es keine Funktionstaste. Bei Auftreten eines Breakpoints oder einer Exception wird die Task automatisch wieder angehalten.

Wenn das Logfile geöffnet und Log verbose gewählt ist, dann wird eine Protokollzeile in das Logfile gegeben.

Eine per Go o.ä. laufende Task kann durch Drücken von <F6> sofort angehalten werden! Allerdings sollten Sie beachten, daß dies mitten im Betriebssystem geschehen kann.

Base register

Enthält die zu debuggende Task Instruktionen, bei denen Daten relativ zu einem Adreßregister adressiert werden, dann kann der Symbolname der Daten für den Disassembler normalerweise nicht angezeigt werden. Mit dieser Funktion legen Sie jedoch ein Adreßregister 0 bis 7 fest, für das die Adressierungsart »offset(An)« durch einen Symbolnamen ersetzt wird. Siehe auch Basereg-Modus des Optimierenden-Makro-Assemblers. Achten Sie darauf, daß Sie beim Assemblieren von Instruktionen über das Disassembler-Fenster evtl. die Adressierung »Absolut lang« statt »offset(An)« wählen. Geben Sie daher bei Programmen mit benutztem Basereg-Modus die Adressierung für solche Befehle immer an, indem Sie den Offset berechnen lassen, z.B. über

```
move symbolname-basisadresse(A5)
```

Base on, Base off

Diese beiden schalten die Offsetdarstellung relativ zu einem Adreßregister ein oder aus. Siehe Basereg-Modus von OMA.

History depth



Die History zeichnet bei Ablauf eines Programmes die abgearbeiteten Befehle auf. Mit dieser Funktion wählen Sie, wieviele Befehle maximal aufgezeichnet werden. Ist die Füllgrenze erreicht, werden die ersten aufgezeichneten Befehle durch neue überlöscht. Sie haben also nur die letzten Befehle zur Verfügung, um einen Fehler, den Sie z.B. gerade beim Single-Step verpaßt haben, doch noch zu lokalisieren.

Beachten Sie, daß die Ausführungszeit bei Go erheblich ansteigt, solange die History eingeschaltet ist.

History on, History off

Die Befehle schalten die History an oder aus. Einschalten kann auch durch die Funktionstaste <F4>, Ausschalten durch <F5> erreicht werden.

### 4.5.4 Das Break-Menü

Set

Setzt einen Breakpoint. Breakpoints verursachen das Anhalten des zu debuggenden Programmes, sobald es auf den Breakpoint stößt. Dann können Sie sich z.B. Variablen angucken und auf ihren Zustand überprüfen. Ist ein Wert angeklickt, wird der Breakpoint auf diesen gesetzt, sonst fragt ein Requester nach der Adresse zum Anhalten. Normale Breakpoints müssen auf geraden Adressen liegen. Gibt es den angegebenen Breakpoint schon, beept das Display einmal.

Set memory

Setzt Speicher-Breakpoints. Diese speziellen Breakpoints ermöglichen Ihnen, das Programm anzuhalten, wenn eine oder mehrere Speicherzellen bestimmte Bedingungen erfüllen.

Auf den Befehl Set memory erscheint ein Requester, in dem eine Adresse eingegeben oder vor Aufruf angeklickt werden kann. Die Bedingung kann Ungleich »!=«, Gleich »==«, Kleiner Gleich »<=« oder Größer Gleich »>=« sein. Ferner müssen der entsprechende Vergleichswert und die Speicherzellengröße (1, 2, oder 4 Bytes) angegeben werden. Beispiel: Es soll angehalten werden, wenn die Wort-Variable »Zaehler« den Wert fünf erreicht hat. Eingabe: Adresse »Zaehler«, Condition »==5«, Size »2«.

Bitte beachten, daß das Programm erheblich langsamer abgearbeitet wird, sofern enablete Speicher-Breakpoints existieren, und daß ein Speicher-Breakpoint nach dem ersten Mal immer wieder auslösen kann, solange die Bedingung zutrifft, also sollte er gleich nach Anhalten gelöscht oder disabled werden.

Clear

Arbeitet wie Set, nur wird ein Breakpoint gelöscht. Gibt es den angegebenen Breakpoint nicht, dann beept das Display einmal.

Enable, Disable

Schalten einen Breakpoint vorübergehend ein bzw. aus. Auch hier erscheint ein Requester, wenn Sie keinen Wert angeklickt haben. Das Ausschalten eines Breakpoints bedeutet nur, daß er das Programm nicht anhält. Sobald er wieder enabled wird, löst er aus. Deswegen können Sie einen Breakpoint setzen, disablen, und das Programm laufen lassen. Während es läuft, können Sie den Breakpoint wieder enablen, und das Programm wird erst danach sofort bei Erreichen des Breakpoints angehalten. Sie können also Routinen austesten, ohne immer wieder anzuhalten, und stoppen nur bei Fehlverhalten.

Das Ein- und Ausschalten läßt sich auch durch Doppelklicken des Enable-Flags im Breakpoint-Fenster bewerkstelligen.

### 4.5.5 Das Command-Menü

Für alle Funktionen wird der gerade angeklickte Wert in das erste Datenfeld übernommen. Alle Funktionen prüfen die Eingabewerte und melden Fehler, falls nötig.

Calculator

Öffnet ein Hilfsfenster zum Ausrechnen von Ausdrücken. Das Ergebnis wird normalerweise als Hexzahl ausgegeben, kann aber in anderen Zahlensystemen ausgegeben werden. Zusammenhang zwischen Ausdrucksform und Ausgabeformat:

**#(ausdruck)** gibt Ergebnis als Dezimalzahl aus

**$(ausdruck)** gibt Ergebnis als Hexadezimalzahl aus

**%(ausdruck)** gibt Ergebnis als Binärzahl aus

Search

Dies öffnet den Such-Requester. Mit diesem können Sie Speicherwerte oder auch Strings suchen. Dazu müssen Sie Start- und Endadresse, den Suchwert, eine Maske (Sie können also auch nach einzelnen Bit-Konstellationen suchen), die Größe des Wertes (1, 2 oder 4 Bytes) sowie die Schrittweite angeben. Geben Sie als erstes Zeichen ein Gleichheitszeichen beim Suchwert an, so können Sie dahinter einen String angeben, nach dem gesucht werden soll. Dieser wird allerdings in Abhängigkeit von Groß- und Kleinschreibung gefunden; Mask-, Step- und Size-Parameter werden dann ignoriert.

Wird eine Stelle gefunden, wird sie ausgegeben, die Startadresse entsprechend weitergestellt und aktiviert. Die Fundadresse können Sie doppelt anklicken und damit ein Speicherfenster öffnen. Das Weitersuchen lösen Sie durch Drücken von Return oder Anklicken von »Ok« aus. Gibt es keine weitere Stelle, meldet der Debugger dies und aktiviert die Endadresse.

Compare

Vergleicht zwei Speicherbereiche miteinander, sobald Sie Startadresse, Endadresse und die Vergleichsadresse angegeben haben. Unterschiedliche Stellen



werden wie bei Search mit ihrer Adresse ausgegeben und haben ebenfalls den Doppelklick zum Öffnen von Speicherfenstern.

Copy

Kopiert einen Speicherbereich an eine andere Stelle. Hier werden Startadresse, Länge und die Zieladresse benötigt.

Fill

Füllt einen Speicherbereich mit einem Wert, der 1, 2 oder 4 Bytes groß sein kann.

Define symbol

Dieser Befehl ist nur bei gewählter Debug-Task erreichbar und definiert ein neues Symbol, welches an der angegebenen Adresse noch nicht existieren darf. Es darf aber ein gleichnamiges Symbol an anderer Stelle geben.

Define hunk

Definiert einen neuen Hunk. Dieser bekommt den Namen »SelfmadeN« mit N als Nummer des N-ten, selbstdefinierten Hunks. Dies bietet Ihnen die schnelle Möglichkeit, während des Debuggens an Zusatzspeicher zu gelangen.

Log data

Hiermit können Sie einen Hex-Dump wie mit einem Speicherfenster in das Logfile geben. Die Daten sind immer ein Langwort groß und haben immer den Textblock. Während des Schreibens ändert sich die Mauszeiger-Gestalt, und der automatische Refresh usw. des *ganzen* Debuggers ist abgeschaltet, damit die File-Operation so schnell wie möglich beendet werden kann.

Log assembly

Arbeitet wie Log data, nur wird ein Disassembler-Listing des angegebenen Bereiches in das Logfile geschrieben.

### 4.5.6 Das Window-Menü

Freeze

Dieser Befehl friert ein Speicher- oder Statusfenster ein. Der Zustand der Fensteranzeige wird dann nicht mehr aktualisiert, d.h. der automatische Refresh wird für dieses Fenster ausgeschaltet. Dies können Sie verwenden, um Änderungen während der Programmausführung zu verfolgen.

Ändern der Fenster-Größeneinstellung oder des Anzeigebereiches bewirken trotz Einfrierens einen einmaligen Anzeigerefresh.

Warm

Taut ein Fenster wieder auf. Es wird sofort auf den neuen Zustand gebracht und unterliegt wieder dem automatischen Refresh.

Update

Dient dazu, eingefrorene Fenster trotzdem zu refreshen. Dieser Refresh erfolgt nur auf Anwählen von Update und z.B. bei Ändern der Fenstergröße. Um z.B. Momentaufnahmen von veränderlichen Speicherzellen zu machen, können Sie das entsprechende Speicherfenster einfrieren und immer wieder Update wählen. Speziell bei schnell veränderlichen Zellen wären Sie sonst nicht in der Lage, die Werte abzulesen.

Duplicate

Dieser Befehl dupliziert das aktuelle Fenster. Eine genaue Kopie des Fensters mit den Einstellungen des alten erscheint exakt über dem alten Fenster. Das neue Fenster bekommt den Namen »Copy of Name«, wobei »Name« der Name des alten Fensters ist.

Retitle

Hiermit können Sie dem aktuellen Fenster über einen Fragerequester einen neuen Titel geben.

Log dump

Dies schreibt den Inhalt des aktuellen Fensters in das Logfile, wenn dieses offen ist. Während des Schreibens ändert sich der Mauszeiger, und der automatische Refresh usw. des ganzen Debuggers ist abgeschaltet.

Front, Back

Diese beiden legen das aktuelle Fenster in den Vorder- bzw. Hintergrund.

Smart refresh?

Ab Kickstart 2.0 ist hiermit wählbar, ob Fenster mit SIMPLE_REFRESH oder SMART_REFRESH laufen. Diese Einstellung wird im durch Save context erzeugten File mit abgespeichert. Der Wechsel kann vollzogen werden, während der Debugger läuft. Dadurch sind gleichzeitig Fenster mit hoher Refreshgeschwindigkeit bzw. mit niedrigem Speicherverbrauch (für Grafikkarten und Großbildschirme) möglich. Das jeweils nächste zu öffnende Fenster bekommt den Refresh, der im Window-Menü eingestellt ist. Leider ist diese Fähigkeit wegen eines 1.3-ROM-Fehlers erst ab 2.0 nutzbar.

Save context

Diese Funktion schreibt die Einstellungen der momentan offenen Debugger-Fenster in ein File namens DBug.prefs in das beim Debugger-Start eingeschaltete Directory. Wenn der Debugger erneut gestartet wird, liest er dieses File und öffnet die Fenster genau so, wie in DBug.prefs abgespeichert. Dies dient dazu, den Debugger für immer wiederkehrendes Debuggen eines Ihrer Projekte weitgehend zu initialisieren.

### 4.5.7 Das Order-Menü

(nur für Symbol-Fenster)



Address, Alphabetic

Nur einer dieser Menüpunkte kann eingeschaltet sein. Sie bestimmen, ob die Symbole eines Symbolfensters alphabetisch oder nach aufsteigender Adresse sortiert sind.

### 4.5.8 Das Source-Menü

(nur für Source-Fenster)

Move with PC, Don't move

Diese beiden Menüs beschreiben im Wechsel, ob der Inhalt des Source-Fensters dem aktuellen PC beim Single-Step folgen soll oder nicht.

Find string

Mit diesem Befehl kann ein String innerhalb des Quelltextes im Source-Fenster gesucht werden. Wenn der String gefunden wird, positioniert der Debugger die entsprechende Zeile als oberste, sichtbare des Source-Fensters. Gesucht wird ab jeweils der zweiten sichtbaren Zeile. Der Vergleich findet ohne Groß-/Kleinschreibung statt.

Tabulator

Hiermit wird der Tabulatorenabstand innerhalb des Quelltextes angewählt, damit Sie die richtige Source-Darstellung erhalten. Defaultmäßig ist dies 8.

### 4.5.9 Das Address-Menü

(nur für Speicherfenster)

Hunk range

Wählt als Speicherfenster-Bereich den Bereich des Hunks, in dem die aktuelle Startadresse liegt. Sofern die Adresse in keinem Hunk vorkommt, wird Physical range genommen.

Physical range

Wählt den physikalischen Prozessorbereich. Dieser ist 0 bis $FFFFFF für MC68000 und MC68010 bzw. 0 bis $FFFFFFFF für MC68020 und höhere Prozessoren, allerdings eingeschränkt durch den tatsächlich vorhandenen Speicher und die Dateien DBug.memory. Siehe auch Abschnitt 4.3.1.

Absolute range

Hier können Sie über Fragerequester einen beliebigen Bereich für das Speicherfenster angeben.

Symbols on, Symbols off

Damit bestimmen Sie, ob Symbolzeilen zwischen die Datenzeilen eingeschoben werden oder nicht.

No offset

Dies druckt die Speicheradresse als hexadezimale Zahl.

Symbol offset

Dies druckt als Adresse den Abstand zum vorigen Symbol im aktuellen Hunk. Gibt es dieses nicht, wird der Abstand zum Bereichsanfang ausgegeben.

Range offset

Dies druckt als Adresse den Abstand zum Bereichsanfang.

Add address, Suppress address

Die beiden Befehle geben an, ob die Speicheradresse als hexadezimale Zahl eingeklammert hinter den Offset gedruckt wird, oder ob sie weggelassen werden soll.

Hexadecimal offset, Decimal offset

Wählt eine hexadezimale bzw. dezimale Zahl für die Offsetdarstellung.

### 4.5.10 Das Data-Menü

(nur für Speicherfenster)

Disassembler

Hiermit schaltet Sie die Darstellung auf den Disassembler. Bitte beachten Sie, daß die übrigen Befehle des Data-Menüs keine Auswirkung haben, solange der Disassembler-Modus eingeschaltet ist. Ausnahme davon sind Unsigned und Signed, die zwischen vorzeichenloser bzw. -behafteter Darstellung bei manchen Befehlen wie MOVE, ADD, SUB, CMP etc. umschalten.

PC-Disassembler

Dies schaltet ebenfalls auf den Disassembler, nur folgt der Fensterinhalt beim Tracen dem aktuellen PC.

Hexadecimal

Wählt die Darstellung mit hexadezimalen Daten.

Decimal

Wählt die Darstellung mit dezimalen Daten.

Binary

Wählt die Darstellung mit binären Daten.

Byte, Word, Long

Diese wählen die Operandengröße.

Unsigned, Signed



Diese drucken die Daten ohne bzw. mit Vorzeichen. Binärzahlen sind immer vorzeichenlos.

Text on, Text off

Bestimmt, ob ein Textblock hinter den Daten folgt.

Auto width, Specify width

Normalerweise kommen so viele Daten in ein Speicherfenster, wie ohne Leseverlust dargestellt werden können (minimal ein Datum); diese Einstellung ist Auto width. Mit Specify width können Sie nun eine feste Anzahl an Daten pro Zeile festlegen. Die Anzahl 0 schaltet Auto width ein.

# Kapitel 5

# DLink V1.2

## 5.1 Einführung

Ein Linker hat die Aufgabe, aus mehreren Modulen bestehende Programmteile miteinander zum fertigen Programm zu verbinden. Dabei können auch in Bibliotheken (Libraries) ausgelagerte Codefragmente in das Programm eingebunden werden.

DLink ist weitgehend kompatibel mit ALink, dem Amiga-Standardlinker. Er verarbeitet Objektfiles und Libraries im Amiga-DOS-Hunkformat und unterstützt die Produktion von sehr kurzen Programmfiles sowie das »Small-Data-Modell« des Optimierenden-Makro-Assemblers. DLink ist ein Linker, der die Objektfiles nur einmal liest und Libraries nur einmal durchsucht. Daraus resultiert eine hohe Geschwindigkeit, wobei der Linkerlauf praktisch nur so lange wie das Laden der Objektfiles und Libraries und das Schreiben des fertigen Programmes dauert.

DLink ist sparsam mit Speicher, aber für größere Programme (größer 60 KByte) muß genügend Speicher verfügbar sein.

Zum Verstehen der genaueren Arbeitsweise von DLink sollte Ihnen das Hunkformat von Amiga-DOS bekannt sein. Ferner empfehle ich, die verschiedenen Optionen des Linkers einfach auszuprobieren, insbesondere Map- und Cross-Referenz-File.

Bisherige Beschränkungen:

- DLink unterstützt keine Overlays. Für solche seltenen Fälle können Sie den Commodore-Linker alink verwenden, der sich im tools-Directory befindet.
- Symbol- oder Hunk- o.ä. Namen dürfen nicht länger als 1020 Zeichen sein.
- DLink leitet keine allgemeinen Debug-Hunks an das Zielfile weiter, da das Format von Debug-Hunks nicht genügend standardisiert ist. DLink unterstützt aber die Debug-Hunks des Optimierenden Makro-Assemblers für Source-Level-Debugging.
- DLink verarbeitet keine »hunk_common« und kennt keine residenten Libraries.

## 5.2 Aufruf des Linkers

DLink kann nur über einen CLI-Aufruf gestartet werden. Die CLI-Parameter gehorchen den CLI-Konventionen, d.h. der String »a**b« ergibt den Filenamen »a*b«. Die Syntax lautet:

```
DLink [FROM|ROOT] Files TO Programm [Hauptoptionen]
```

Files sind Objektfiles, die in jedem Fall in das Zielprogramm kommen. Diese Filenamen dürfen durch *FROM* oder *ROOT* eingeleitet werden und können durch Leerzeichen, Pluszeichen oder Kommata getrennt werden. Soll ein Filename Leerzeichen, Pluszeichen oder Kommata enthalten, muß er alleine und in doppelte Hochkommata eingefaßt stehen. Filenamen, die nicht durch eine der folgenden Hauptoptionen eingeleitet werden, gelten auch als Objektfiles. Es muß mindestens ein Objektfile angegeben werden; die Objektfiles werden in der Reihenfolge in das Programm gebunden, in der sie angegeben werden. Deswegen sollten Sie das Objektfile mit der ersten auszuführenden Instruktion des Programmes als erstes angeben, da sonst evtl. die Ausführung in z.B. einer Data-Sektion beginnt. Es können beliebig viele Objektfiles angegeben werden (auch durch das Withfile, s.u.).

Die *TO*-Option mit dem darauf folgenden Programmnamen muß angegeben werden, damit der Linker weiß, welches Programm erzeugt werden soll.

Gültige Hauptoptionen sind:

**WITH Withfile**

Mit dieser Option werden weitere Optionen aus dem Textfile Withfile gelesen, welche mit den Optionen der CLI-Aufrufzeile gemischt werden. Es ist nicht möglich, z.B. die VER-Option doppelt vorkommen zu lassen (einmal in der Aufrufzeile, einmal im Withfile). Nur Libraries dürfen mehrfach vorkommen, und es dürfen mehrere Objektfiles angegeben werden, wobei aber kein Name mehrfach erlaubt ist. Eine Withfile-Zeile darf maximal 254 Zeichen lang sein, das Withfile hat aber keine Größenbeschränkung. Ein Withfile hat folgende Syntax, wobei einige Optionen weggelassen werden dürfen, die WITH-Option nicht erlaubt ist und auch mehrere Optionen in einer Zeile stehen dürfen:

```
[FROM|ROOT] files
TO Programm
VER file
LIB library
MAP file
XREF file
WIDTH n
SMALL
STAT
STRIP
SHORT
```

**VER Meldungsfile**



Normalerweise gehen alle Meldungen des Linkers auf das Terminal (das aktuelle CLI-Fenster). Mit dieser Option werden die Textausgaben auf das File Meldungsfile umgeleitet.

**LIBRARY Library[(","|"+")Library2]...**
**LIB Library**

Es werden Libraries angegeben, die auf Werte durchsucht werden, die nicht in den Objektfiles definiert wurden. Findet DLink in der Library ein verlangtes Symbol, wird die entsprechende »hunk_unit« in das fertige Programm einbezogen und so die Referenz aufgelöst. Es können mehrere Libraries durch Pluszeichen oder Kommata getrennt angegeben werden. Soll ein Libraryname Pluszeichen oder Kommata enthalten, muß er alleine und in doppelte Hochkommata eingefaßt stehen. Die LIB-Option darf mehrfach vorkommen. Die Libraries werden in der Reihenfolge durchsucht, in der sie angegeben sind. Dies ist wichtig für Libraries, die dieselben Routinennamen verwenden, bei denen die Routinen aber unterschiedliche Fähigkeiten haben.

Es können beliebig viele Libraries angegeben werden. Eine CLI-Zeile darf zwar nicht länger als 255 Zeichen sein, aber per Withfile werden Ihnen keine Einschränkungen auferlegt.

**MAP Mapfile**

Mit dieser Option produzieren Sie am Ende des Linkerlaufes ein Textfile namens Mapfile, das für jede eingebundene Sektion Informationen darüber enthält, welche Symbole die Sektion für den Linker sichtbar gemacht hat und welchen Wert jedes Symbol hat. Absolute Symbole sind mit einem Stern gekennzeichnet.

Obwohl Symbolnamen bis zu 1020 Zeichen lang sein dürfen, werden im Map- und auch im Cross-Referenz-File maximal 32 Zeichen ausgegeben.

**XREF Crossreferenzfile**

Diese Option funktioniert ähnlich wie MAP, nur wird hier statt des Symbolwertes für jede Sektion und jedes Symbol aufgeführt, wo es von anderen Sektionen verlangt wurde. Diese Information umfaßt die Hunknummer innerhalb des fertigen Programmes und den Offset innerhalb dieses Hunks. Mittels der Cross-Referenz können z.B. Symbole erkannt werden, die überhaupt nicht benutzt werden.

**WIDTH n**

Der Wert legt für das XREF-File fest, wieviel Zeichen eine Zeile maximal enthalten darf. Der Wert *n* muß im Bereich 50 bis 255 liegen.

**SMALL**

Schaltet den Small-Modus ein (*nicht* dasselbe wie das Small-Data-Modell). Damit kommen alle Code-Hunks in einen Code-Hunk, ebenso alle Data-Hunks in einen Data-Hunk und alle Bss-Hunks in einen Bss-Hunk unabhängig vom durch »hunk_name« definierten Namen. Es werden nur Hunks mit gleichem Speichertyp vereinigt; z.B. wird ein Code-Hunk mit PUBLIC-Speichertyp nicht mit einem Code-Hunk mit CHIP-Speichertyp zusammengepackt.

**STAT**

Gibt nach dem Linkerlauf eine Statistik aus. Diese umfaßt die für den Lauf benötigte Zeit und den maximalen Speicherbedarf.

**STRIP**

Mit dieser Option erzeugt DLink das fertige Programm ohne die Symboltabelle (ohne »hunk_symbol«). Debug-Hunks außer denen, die OMA selber erzeugt, erscheinen grundsätzlich nicht im Zielprogramm, da deren Format nicht genormt ist und bei verschiedenen codeerzeugenden Programmen ganz anders aussehen kann.

**SHORT**

Diese Option ermöglicht das »Small-Data-Modell« für in Module getrennte Programme. Siehe Kapitel »Short-Modus«.

**NOCASE**

Diese Option bewirkt, daß DLink bei Symbolnamen keine Unterscheidung zwischen Groß- und Kleinschreibung macht. Dies werden Sie benötigen, wenn Sie die »-c c«-Startoption des Assemblers benutzen, da sonst Namen in Libraries nicht mit den angeforderten Symbolnamen übereinstimmen.

Der Linker kann jederzeit mit <Ctrl>+<C> angehalten werden. Die Fehlermeldungen von DLink sind präzise und bedürfen keiner Erläuterung. Eine Ausnahme bildet der Fehler »Inkompatible Körpertypen«, der dann gemeldet wird, wenn bei zwei verschiedenen Sektionen die Namen übereinstimmten, der Speichertyp und/oder der Sektionstyp aber nicht übereinstimmten.

Beispielaufrufe:

```
DLink a.obj b.obj TO prg
DLink a.obj+b.obj to prg
DLink a.obj,b.obj to prg
```

Linkt a.obj und b.obj zu dem Programm prg.

```
DLink WITH Withfile
```

Und Withfile enthält

```
a.obj b.obj        (Zeile 1)
TO prg             (Zeile 2)
```

Linkt a.obj und b.obj zu dem Programm prg.

```
DLink a.obj+b.obj "c.obj" to "Pro 1"
```

Linkt a.obj, b.obj und c.obj zu dem Programm Pro 1.

```
DLink a.obj b.obj LIB amiga.lib TO prg STRIP
```

Linkt a.obj und b.obj zu dem Programm prg, holt evtl. Teile aus der Library amiga.lib und verhindert, daß prg eine Symboltabelle hat.



```
DLink a.obj,b.obj lib amiga.lib+debug.lib to prg strip
DLink a.obj,b.obj lib amiga.lib library "debug.lib" to prg strip
```

Wie voriges Beispiel, nur wird die Library debug.lib auch durchsucht.

```
dlink a.obj+b.obj to prg map *
```

Linkt und erzeugt ein Mapfile, welches auf das Terminal geht.

## 5.3 Short-Modus

Zur Verdeutlichung der Problematik eines »Small-Data-Modell«s bei getrennten Modulen sei hier auf das Kapitel »Verkürzte Adressierung« des Assembler-Handbuches verwiesen.

Normale Linker können keine 16-Bit-Referenz auf ein Symbol lösen, das nicht zur gleichen Sektion gehört, und erzeugen eine Fehlermeldung. Mit der SHORT-Option des DLink wird korrekt gelinkt, falls der resultierende 16-Bit-Wert nicht außerhalb seines Bereiches gerät und das Symbol zu einer Bss-Sektion gehört. DLink sorgt automatisch für einen Offset für eine 64 KByte große Datensektion, indem er 32768 von dem Offset abzieht. Es wird also immer angenommen, daß der Basiszeiger auf den Beginn der Datensektion plus 32768 zeigt.

Der Short-Modus funktioniert auch, wenn mehrere Objektfiles Teile ein und derselben Bss-Sektion enthalten, solange die Basisadresse in dem Objektfile definiert wird, das als erstes von diesen Objektfiles in der Aufrufzeile steht. Für das folgende Beispiel lautet der Linkeraufruf

```
DLink a.obj b.obj c.obj to prg SHORT
```

File a.asm:

```
      CODE
      XREF  basis,wert1,wert2
      lea   basis+32768,a5       ; Initialisierung des Basiszeigers
      move  d0,wert1(a5)         ; wird zu MOVE D0,-32764(A5)
      move  d0,wert2(a5)         ; wird zu MOVE D0,-32760(A5)
      rts
      END
```

File b.asm:

```
      BSS
      XDEF  basis,wert1
basis ds.l  1
wert1 ds.l  1
      END
```

File c.asm:

```
      BSS
      XDEF  wert2
wert2 ds.l  1
      END
```

## 5.4 Libraries

Libraries sind Sammlungen von Objektfiles. Sie werden mit der LIB-Option des Linkers durchsucht, und DLink holt nur solche Objektfiles aus der Library, die Definitionen enthalten, die noch im Linkerlauf benötigt werden. Diese Definitionen können Routinen oder absolute Konstanten sein.

Sie können eigene Libraries herstellen, indem Sie ganz normale Objektfiles mit Konstanten-Definitionen und/oder Routinen per Assembler erstellen, die jeweils mindestens ein Symbol mit der Assembler-Direktive XDEF für den Linker sichtbar machen. Diese Objektfiles brauchen Sie dann nur mit dem CLI-Befehl *JOIN* zu einem File, der Library, zu vereinigen. Eine Library kann aus nur einem Objektfile bestehen.

Wenn Sie Konstanten oder Routinen aus einer Library in ihre Programme einbinden möchten, geben Sie die Library beim Linkeraufruf mit der LIB-Option an. Der Linker wird dann nur die Objektfile-Teile aus der Library in das fertige Programm übernehmen, die Symbole definieren, welche Sie in ihren Programmen mit der Assembler-Direktive XREF verlangen.

Da eine Library von vorn bis hinten durchsucht wird, sollten Sie dafür sorgen, daß Library-Teile, die Konstanten aus derselben Library benötigen, so weit wie möglich am Anfang der Library stehen. Sonst kann folgendes passieren: Das Symbol »Multi« wird in der Library definiert, der Linker beachtet dieses Symbol aber vorerst nicht, weil es noch nicht verlangt wurde. Weiter hinten in der Library steht eine Routine, die verlangt wird und die selber das Symbol »Multi« benötigt. Diese Routine wird eingebunden, und das Symbol »Multi« wird als benötigt vermerkt. Da der Linker aber über die Definitionsstelle von »Multi« hinaus ist, wird er einen Fehler erzeugen, da er »Multi« nicht mehr findet. Einzige Abhilfe wäre das zweimalige Durchsuchen der Library durch zweimaliges Angeben per LIB-Option (was aber Zeit kostet). Deswegen sollten Sie beim Aufbau ihrer eigenen Libraries auf diese Problematik achten.

Im Examples-Directory der Originaldiskette finden Sie den kompletten Quelltext der mitgelieferten, »amiga.lib«-kompatiblen Library »OMA.lib« mitsamt den nötigen Makefiles. Hier sehen Sie sehr genau, wie eine Library aufgebaut werden kann und wie sie mittels »Make« aktualisiert gehalten wird. Wenn Sie diese Library abändern möchten, dann kopieren Sie die Quelltexte mitsamt Unterdirectories auf eine freie Diskette und fügen die notwendigen OBJ-Unterdirectories hinzu. Danach brauchen Sie nur noch im Haupt-Directory der Library zu sein und »make« aufzurufen (arbeiten Sie *nicht* mit der Originaldiskette), um zu der geänderten Version zu kommen.

Wenn Sie eine Library als Objektfile beim Linken angeben, werden alle Teile der Library in das erzeugte Programm eingebunden. Dies können Sie z.B. nutzen, indem Sie mehrere Objektfiles per *JOIN* zusammenhängen und später als ein einziges File beim Linken benutzen.

# Kapitel 6

# Hilfsprogramme und die Library OMA.lib

Sämtliche hier dokumentierten Hilfsprogramme können Sie nur vom CLI aus benutzen. Für jedes Hilfsprogramm wird die CLI-Startsyntax genau beschrieben.

## 6.1 Make

*Make* ist ein Make-Utility, mit dem Programmpakete halbautomatisch auf dem neuesten Stand gehalten werden können, solange die Systemuhr richtig eingestellt ist. Die Funktion des Make-Utilities ist im Handbuch zum Editor beschrieben, bitte lesen Sie dort nach. Die CLI-Syntax lautet:

```
MAKE [Filename]
```

Filename ist ein optionaler Name für das dem Makelauf zugrunde liegende Makefile. Lassen Sie Filename weg, dann benutzt *Make* den Namen makefile. Im Gegensatz zum im Editor integrierten Make-Utility kann das Makefile mit dem eigenständigen *Make*-Hilfsprogramm also einen anderen Namen als makefile haben. Zudem läßt sich *Make* dann auch mit kleinen 512-KByte-Hauptspeichern betreiben. Beispiel-Makefiles finden Sie im Examples-Directory.

## 6.2 Absolute

*Absolute* ist ein Hilfsprogramm, das aus einem ausführbaren Amiga-DOS-Programm ein absolutes Programm erzeugt, welches Sie z.B. in ein EPROM brennen können. Die CLI-Syntax lautet:

```
ABSOLUTE Quellfile [TO] Zielfile $Adresse
ABSOLUTE Quellfile QUERY
```

Quellfile ist das ausführbare Amiga-DOS-Programm, das umgewandelt werden soll, und Zielfile das von *Absolute* erzeugte EPROM-fähige Programm. Der Parameter *TO* vor Zielfile ist optional, darf also weggelassen werden. *$Adresse* ist die Startadresse für das EPROM-Programm, z.B. *$FC0000* für den Amiga-ROM-Beginn. Bei der zweiten Syntax werden Sie für jeden Hunk gefragt, in welches Zielfile er gelangen soll und welche Startadresse für diesen Hunk benutzt werden soll. Damit ist es möglich, ein Programm an mehrere verschiedene Speicherstellen zu legen. Beachten Sie, daß jeder Hunk damit in ein eigenes File gelangt, damit er an seine Zieladresse geladen werden kann. Sie können *Absolute* auch dazu mißbrauchen, vom Assembler erzeugte Binärtabellen aus dem Amiga-DOS-Hunkformat herauszutrennen und per INCBIN-Direktive in andere Quelltexte einzuschließen. Dazu sollten die Tabellen aber nur absolute Daten enthalten, da die Relokationsinformation verloren geht.

## 6.3 Strip

*Strip* ist ein Programm, das aus Objektfiles, Libraries oder fertigen Programmen die Debug-Hunks und Symboltabellen entfernt. Die CLI-Syntax lautet:

```
STRIP Quellfile [TO] Zielfile
```

Quellfile ist das Objektfile, die Library oder das fertige Programm, und Zielfile ist das von *Strip* erzeugte File ohne die Debug-Hunks und die Symboltabellen. *TO* ist optional.

Für *Strip* ist der Assembler-Quelltext im Examples-Directory unter dem Namen Strip.asm enthalten. Das Makefile für *Strip* steht dort unter dem Namen makefile.

## 6.4 Cmp

*Cmp* vergleicht zwei Binärfiles miteinander und meldet, ob und wo sie verschieden sind. Die CLI-Syntax lautet:

```
Cmp File1 File2
```

Wenn beide Files gleich sind, wird nichts gemeldet. Der Assembler-Quelltext für *Cmp* ist im Examples-Directory unter dem Namen Cmp.asm enthalten.

## 6.5 CLS

*CLS* löscht das aktuelle CLI-Fenster. Der Assembler-Quelltext ist im Examples-Directory unter dem Namen CLS.asm enthalten.



## 6.6 Das Include-Directory

Das Directory Include enthält die Amiga-Includefiles 2.04, Version 37.4. Zusätzlich zu den Commodore-Standardincludefiles gibt es Includefiles im lvo-Directory, die die Offset-Definitionen für die Standard-Systemlibraries enthalten. Eine Neuerung: exec/types.i enthält jetzt Makros, die automatisch bei z.B. LONG auf eine Wortgrenze gehen.

Die von Commodore original herausgegebenen Includefiles enthielten einige Fehler und wurden fehlerbereinigt, die Änderungen finden Sie in der Datei LiesMich. Damit Sie beim Assemblieren eine hohe Geschwindigkeit erreichen, wurden in den Dateien in Include die meisten Kommentare entfernt, und sie sind speziell an OMA angepaßt. Da die originalen Commodore-2.04-Includefiles für die Programmierung wichtige Kommentare enthalten, wurde die Version mit Kommentaren gepackt beigelegt.

Include1.3 enthält die alten Amiga-Includefiles der Workbench 1.3. Das Symbol ≫TC≪ wurde in exec/tasks.i in ≫TC_≪ geändert, um die Namenskollision mit dem PMMU-Register zu vermeiden.

## 6.7 Das Examples-Directory

Examples enthält Quelltexte für einige der Programme und Beispiele der Assembler-Programmierung und in diesen auch deren Beschreibung. Im Unterdirectory Library ist der komplette Quelltext für die ≫amiga.lib≪-kompatible Library ≫OMA.lib≪ enthalten. ≫OMA.lib≪ selber finden Sie im Libraries-Directory. Im Unterdirectory Startup ist der Quelltext für die Startup-Objektfiles aus dem Libraries-Directory enthalten.

## 6.8 Das Libraries-Directory und OMA.lib

Dieses Directory enthält die Library OMA.lib sowie zwei Startup-Objektfiles, nämlich startup.obj für den kombinierten CLI- bzw. Workbench-Startup und cli.obj nur für das CLI. Die Quelltexte zu diesen Dateien befinden sich im Directory Examples/Startup bzw. Examples/Library.

Die OMA.lib ist eine zur ≫amiga.lib≪ kompatible Library. Für alle System-Bibliotheken des Amiga sind C-Interfaces vorhanden, die die Parameter rückwärts auf den Stack geschoben erwarten. Die IEEE-Trans-Library der Workbench 1.3 wird auch unterstützt. Ferner sind Hilfsroutinen vorhanden, deren Beschreibung unten folgt. Beachten Sie, daß die Routinennamen für den Assembleraufruf mit einem Unterstrich beginnen müssen. Die Quelltexte zur OMA.lib sind vollständig tabulator-komprimiert vorhanden.

Die Unterstützungsroutinen namens _AddICRVector, _RemICRVector, _SetICR und _AbleICR wird es nicht geben, ebenso fehlen alle Routinen der Form faddi, fmuli usw. Benutzen Sie dafür von Assembler aus die direkten Aufrufe per ≫jsr _LVOabc(a6)≪ oder die C-Interface-Routine ≫jsr _abc≪.

amiga.lib ist neben ein paar anderen originalen Commodore-Bibliotheken im Directory Libs2.0 enthalten. Die Startup-Programme von Commodore finden sich im Directory startups.

---

`NewList (Zeiger auf Listenstruktur)`

Ermöglicht es, den NEWLIST-Makro von C aus zu nutzen.

`BeginIO (IO-Zeiger)`

Ermöglicht es, den BEGINIO-Makro von C aus zu nutzen.

`Portzeiger = CreatePort (Namenzeiger, Priorität)`

Erzeugt einen Port mit gegebenem Namen und Priorität. Rückgabewert in D0; gibt Null zurück, wenn Port nicht erzeugt werden konnte.

`DeletePort (Portzeiger)`

Entfernt den Port aus dem System.

`IOStdReq-Zeiger = CreateStdIO (ioReplyPort)`

Erzeugt eine Standard-IO-Struktur. »ioReplyPort« zeigt auf den Antwort-Port. Rückgabe in D0; gibt Null zurück, wenn etwas schiefging.

`DeleteStdIO (IOStdReq-Zeiger)`

Gibt die erzeugte Struktur wieder frei.

`IORequest-Zeiger = CreateExtIO (ioReplyPort, Requestgröße)`

Erzeugt eine erweitere Standard-IO-Struktur. »Requestgröße« gibt deren Größe an und sollte nie kleiner als IOSTD_SIZE sein! Rückgabe in D0; gibt Null zurück, wenn etwas schiefging.

`DeleteExtIO (IOExt-Req-Zeiger)`

Gibt die erzeugte Struktur wieder frei.

`Taskzeiger = CreateTask (Namenzeiger, Priorität, AnfangsPC, Stackgröße)`

Erzeuge eine Task. Rückgabe in D0; gibt im Fehlerfall Null zurück.

`DeleteTask (Taskzeiger)`

Entfernt die Task aus dem System.

---

`IDivS (arg1, arg2)`

Teilt arg1 (32 Bit) durch arg2 (16 Bit) vorzeichenbehaftet. Rückgabe: 32-Bit-Ergebnis vorzeichenbehaftet.

`IDivU (arg1, arg2)`

Teilt arg1 (32 Bit) durch arg2 (16 Bit) vorzeichenlos. Rückgabe: 32-Bit-Ergebnis vorzeichenlos.



`IMulS (arg1, arg2)`

Multipliziert arg1 (16 Bit) mit arg2 (16 Bit) vorzeichenbehaftet. Rückgabe: 32-Bit-Ergebnis vorzeichenbehaftet.

`IMulU (arg1, arg2)`

Multipliziert arg1 (16 Bit) mit arg2 (16 Bit) vorzeichenlos. Rückgabe: 32-Bit-Ergebnis vorzeichenlos.

Die folgenden Routinen gibt es nur für Assembler und *ohne den Unterstrich.* Sie arbeiten mit reiner 32-Bit-Arithmetik.

`lmult ulmult`

D0 = D0 * D1 vorzeichenlos oder vorzeichenbehaftet.

`ldivt`

D0 = D0 / D1 vorzeichenbehaftet.

`lmodt`

D0 = D0 mod D1 vorzeichenbehaftet.

`uldivt`

D0 = D0 / D1 vorzeichenlos. Zusätzlich: Modulo in D1.

`ulmodt`

D0 = D0 mod D1 vorzeichenlos. Zusätzlich: Quotient in D1.

`FFP = afp (Stringzeiger)`

Wandelt einen ASCII-String in eine FFP-Fließkommazahl um. Rückgabe in D0.

`FFPAFP`

Assemblerversion von _afp *ohne* den Unterstrich im Namen. Eingabe: A0 zeigt auf den umzuwandelnden ASCII-String. Ausgabe: D7 und D0 enthalten die FFP-Fließkommazahl und A0 zeigt direkt hinter den gelesenen ASCII-String. Falls ein Fehler auftrat, ist das Carry-Flag gesetzt, und A0 zeigt auf die Fehlerstelle.

`Exponent = fpa (FFP-Zahl, Stringzeiger)`

Wandelt eine FFP-Fließkommazahl in einen ASCII-String um.

`FFPFPA`

Assemblerversion von _fpa. Eingabe: D0 enthält die FFP-Fließkommazahl und A0 zeigt auf den Zielstring. Ausgabe: D0 enthält den Exponenten als vorzeichenbehafteten 32-Bit-Integer.

---

`FastRand (Zahl)`

Erzeugt schnell einen Pseudo-Zufallszahl unter Benutzung der »Zahl«. Rückgabe: Die Zufallszahl. Sollte für das nächste Mal gespeichert werden.

`RangeRand (Bereich)`

Erzeugt eine Zufallszahl im Bereich 1 bis ≫Bereich≪. ≫Bereich≪ muß im Bereich 1 bis 65535 liegen. Rückgabe: Die Zufallszahl.

`AddTOF (Interruptstruktur-Zeiger, C-Routinen-Zeiger, Datenfeldzeiger)`

Hängt eine C-Routine in die Vertical-Blank-Interrupt-Serverkette. Der C-Routine wird der Zeiger auf das Datenfeld übergeben. Rückgabe: Keine. Siehe auch den Quelltext.

`RemTOF (Interruptstruktur-Zeiger)`

Entfernt die C-Routine wieder aus der Serverkette. Rückgabe: Keine.

`waitbeam (Positionszeile)`

Wartet, bis der Videostrahl über die Positionszeile gelaufen ist. Rückgabe: Die aktuelle Videostrahl-Position.

---

`puts (Stringzeiger)`

Schreibt einen String auf ≫stdout≪. Das Stringende 0 wird durch ein Linefeed ersetzt. Rückgabe: Linefeed. Im Fehlerfall -1.

`fputs (Stringzeiger, File)`

Schreibt einen String in ein File. Das Stringende 0 wird nicht geschrieben. Rückgabe: Das letzte ausgegebene Zeichen. Im Fehlerfall -1.

`getchar ()`

Lies genau ein Zeichen von ≫stdin≪. Rückgabe: Das gelesene Zeichen. Im Fehlerfall oder bei Fileende -1.

`putchar (Zeichen)`

Gib ein Zeichen auf ≫stdout≪ aus. Rückgabe: Das ausgegebene Zeichen. Im Fehlerfall -1.

`fgetc (File)`

Lies genau ein Zeichen aus einem File. Rückgabe: Das gelesene Zeichen. Im Fehlerfall oder bei Fileende -1.

`fputc (Zeichen, File)`

Gib ein Zeichen auf ein File aus. Rückgabe: Das ausgegebene Zeichen. Im Fehlerfall -1.

`fclose (File)`

Schließe das File.

---

`sprintf (Stringzeiger, Formatstringzeiger {Argliste)}`

Formatiert den Formatstring mit der Argumentliste und schreibt den Zielstring in (Stringzeiger). Rückgabe: Der Zeiger auf den Zielstring.



`fprintf (File, Formatstringzeiger {Argliste)}`

Formatiert den Formatstring mit der Argumentliste und schreibt den Zielstring in ein File. Rückgabe: 0. Im Fehlerfall -1.

`printf (Formatstringzeiger {Argliste)}`

Formatiert den Formatstring mit der Argumentliste und schreibt den Zielstring nach ≫stdout≪. Rückgabe: 0. Im Fehlerfall -1.

# Kapitel 7

# Objektorientierte Programmierung

## 7.1 Einführung in OOP

Dieser Abschnitt ist eine kurze Einführung in die allgemeinen Begriffe der objektorientierten Programmierung. Es ist ratsam, zu dem Thema bereits Vorkenntnisse zu besitzen oder Sekundärliteratur zu lesen.

### 7.1.1 Motivationsbeispiel

Normale Programmentwicklung sollte mit dem Modulkonzept arbeiten, bei dem ein Aufgabengebiet einem Modul zugeteilt wird. Z.B. wird für ein Anwendungsprogramm ein Fenstermodul entworfen, das sich um die Verwaltung eines Fensters kümmert. Zu seinen Aufgaben gehört das Öffnen eines Fensters, das Schließen, das Ändern der IDCMP-Flags sowie das Anhängen und Entfernen eines Menüs. Damit mehrere Fenster verwaltet werden können, hat das Modul noch eine Routine zum Erzeugen einer Fensterverwaltungs-Struktur und zum Löschen derselben. Das Modul wird implementiert und funktioniert zufriedenstellend. Nun bemerkt man, daß das Programm neben einfachen Fenstern mit Textdarstellung noch solche benötigt, die irgendwelche Gadgets enthalten. Natürlich kann das Fenstermodul entsprechend erweitert werden, aber ab einer bestimmten Anzahl verschiedener Fenstertypen hat man im Modul eine Menge Abfragen, welcher Typ denn gerade vorliegt und welche Aktionen daraufhin ausgeführt bzw. weggelassen werden müssen. Je mehr neue Typen hinzugefügt werden, desto undurchsichtiger wird die Situation, und die Fehlerwahrscheinlichkeit steigt zusammen mit der Verzweiflung.

Die objektorientierte Programmierung (*OOP*) ist hier in der Lage, die Behandlung der verschiedenen Fenstertypen auseinanderzuziehen und voneinander abzugrenzen.

### 7.1.2 OOP-Grundbegriffe

OOP faßt die Welt als Ansammlung von Objekten auf. Ein *Objekt* ist dabei das Modell eines realen (oder nur im Rechner), existierenden Gegenstandes mit seinen spezifischen und allgemeinen Eigenschaften. Ein Objekt wird im Rechner als Datencontainer zusammen mit Routinen, die das Objektverhalten bestimmen, repräsentiert, d.h., ein Objekt besteht aus seinen Daten und den zugehörigen Routinen. Ein Mensch hat z.B. als Daten eine Altersangabe, ein Geschlecht und einen Namen, und als Routinen das Laufen, Sitzen, Altern usw. Da es meist viele Objekte mit gleichem Datenaufbau und gleichen Routinen gibt, werden alle Objekte mit den gleichen Eigenschaften derselben *Klasse* zugeteilt. Z.B. haben alle Objekte der Klasse »Mensch« das Laufen, Sitzen und Altern gemeinsam, und es wäre Speicherverschwendung, für jeden einzelnen Menschen diese Routinen zu implementieren. Neben den Routinen, die das Verhalten der Objekte realisieren, enthält die Klasse noch die Routinen zum Erzeugen und Löschen von Objekten. Alle Routinen werden als *Methoden* bezeichnet (warum, wird unten klar).

Die Klasse ist somit nur eine Schablone für Objekte; sie bestimmt die Struktur sowie die Methoden der Objekte. Während der Laufzeit eines OO-Programmes müssen erst noch Objekte dieser Klassen erzeugt werden. Ein erzeugtes Objekt wird als Objekt-*Instanz* bezeichnet. Es besteht entsprechend der Schablone aus den Daten, die jedes Objekt nur für sich hat, z.B. das Alter eines Menschen, welches von Menschobjekt zu Menschobjekt meist verschieden ist. Diese Daten heißen *Instanzvariablen*. Zusätzlich enthält es einen Verweis auf die Klasse, der es angehört. Die Klasse wird durch den *Klassencontainer* repräsentiert, der aus ein paar Daten wie z.B. der Instanzgröße und dem Code der Methoden besteht.

Eine fertige Klasse kann erweitert werden, ohne daß sie geändert zu werden braucht. Dabei entsteht eine neue Klasse, die weitgehend das Verhalten der alten Klasse besitzt, aber deren Funktionalität ändern kann und neue Funktionalität hinzufügen darf. Diesen Vorgang bezeichnet man als *Erben*. OMA unterstützt nur das einfache Erben, bei dem genau eine Klasse als Vererber erlaubt ist. Die Erbrelation ergibt eine *Vererbungshierarchie*, in der es vererbende Klassen, *Oberklassen* oder *Superklassen*, gibt, und erbende *Unterklassen* bzw. *Subklassen*. Sollte Klasse C von Klasse B erben, die wiederum von Klasse A erbt, so ist auch A eine Oberklasse von C, da C über B Eigenschaften von A erbt. B hat aber von allen Oberklassen von C eine Sonderstellung, da es die *direkte Oberklasse* von C ist. A ist nur eine *indirekte Oberklasse* von C, aber Bs direkte Oberklasse.

Im obigen Beispiel gäbe es eine Klasse `Fenster`, von der sowohl die Klasse `Textfenster` als auch die Klasse `Gadgetfenster` erben würden. In `Fenster` käme all die Funktionalität, die `Textfenster` und `Gadgetfenster` vollkommen gemeinsam haben. In den Unterklassen findet sich dann diejenige, die für die Klasse spezifisch ist, z.B. die Gadgetverwaltung ausschließlich in `Gadgetfenster`, da unsere `Textfenster` keine Gadgets besitzen sollen. Da ein Objekt der Klasse `Fenster` keinen Sinn macht (sie hat nicht genügend Funktionalität), sondern erst ein Objekt einer von ihr erbenden Klasse, faßt `Fenster` nur gemeinsame Konzepte zusammen. `Fenster` kann auch Methoden deklarieren, die erst in den Unterklassen existieren. Eine solche Klasse wie `Fenster` heißt eine *abstrakte Klasse*. Sie legt nur fest, welches Verhalten in den Unterklassen existieren muß, und wird selber niemals Objekte erzeugen.



Durch die Erbhierarchie mit Klassen werden zwei Relationen zwischen Objekten und Klassen erzeugt. Die eine Relation ist die »ist-ein«-Relation, z.B. »ist-ein« Objekt der Klasse `Gadgetfenster` ein Gadgetfenster. Die zweite Relation ist die schwächere »ist-eine-Art«-Relation, z.B. ist dieses Objekt gleichzeitig nur eine Art `Fenster`.

### 7.1.3 Methoden

Um mit einem Objekt etwas anfangen zu können, muß man es manipulieren können. Da Objekte Daten abkapseln sollen und daher kein Zugriff auf Instanzvariablen von außerhalb der Klassen erfolgen darf, existieren wie oben gesagt Routinen, über die sämtliche Interaktion mit dem Objekt stattfindet. Diese *Methoden* sind prinzipiell genau so wie Unterroutinen, nur wird die Entscheidung, wohin der Sprung erfolgt, erst zur Laufzeit getroffen. Wenn z.B. Klasse `Hund` die Methode `Hund_Laufen` besitzt und die Klasse `Kuvasz` (eine Hunderasse) von `Hund` erbt und wegen der manchmal eigenartigen Gangweise eines Kuvasz die Methode `Hund_Laufen` ebenfalls implementiert, nur in leicht anderer Weise, so spricht man vom *Überschreiben* der Methode (engl. *overwrite*). Beim Aufruf der Methode `Hund_Laufen` hängt es von der Klassenzugehörigkeit des Objektes ab, welcher Code angesprungen wird. Diese Eigenschaft heißt *Polymorphismus*.

Innerhalb der überschreibenden Methode muß es eine Möglichkeit geben, auch den Code der gleichnamigen Routine in der Oberklasse zu erreichen. Der Aufruf der Methode selber würde nämlich nur zum erneuten Aufruf der Methode derselben Klassenebene und damit wahrscheinlich zu einer Endlosschleife führen. Dies geschieht unter Verwendung des sogenannten *Super*-Aufrufes.

In jedem Fall muß der Methodencode Zugriff auf das Objekt selber haben. Dazu wird implizit jeder Methode der Zeiger auf das Objekt mitgegeben. Dieser Zeiger wird meistens mit *Self* bezeichnet.

Damit die Entscheidung des Sprungziels effizient getroffen werden kann, wird in der Regel der Methodenname bei der Übersetzung des Quelltextes in einen Zahlencode überführt, der *Methoden-ID*.

Anhand der Methoden wird auch deutlich, welche Vorteile OOP bietet. Wenn die Unterklasse für eine bestimmte Methode keine andere Funktionalität benötigt als die Oberklasse, kann sie die Methode einfach miterben, indem sie sie nicht überschreibt. Der Code für Methode ist damit nur einmal, nämlich in der Oberklasse, vorhanden, und die neue Klasse braucht sich nur um neu eingeführte und überschriebene Methoden zu kümmern. Ihr Aufgabengebiet ist somit klar eingegrenzt und übersichtlich.

## 7.2 OOP-Struktur bei OMA

OMA unterstützt auch das objektorientierte Programmieren. Damit die Nutzung von OOP unter OMA recht bequem gestaltet ist, gibt es das Includefile `ooma.i`, in dem Makros und Strukturen zur einfachen Deklaration von Klassen, Methoden und Objekten

enthalten sind. Die entsprechenden Routinen müssen mit dem Modul ooma.obj hinzugelinkt werden. Bevor irgendeine andere Aktion der Klassen- und Objektverwaltung angefordert werden darf, muß `InitObjectModule` aufgerufen worden sein.

### 7.2.1 Klassenhierarchie und ClassObject

An der Spitze der Klassenhierarchie von OMA steht die abstrakte Klasse `ClassObject`, die direkte oder indirekte Oberklasse von allen anderen Klassen sein muß. Die Basisklasse `ClassObject` definiert den Verweis auf den Klassencontainer sowie die Methoden `Object_Initialize`, `Object_Cleanup` und `Object_Delete`. `Object_Initialize` muß in den Unterklassen realisiert werden und sollte dort das Objekt in einen definierten Zustand bringen. `Object_Cleanup` sollte kurz vor dem Löschen des Objektes aufgerufen werden, damit z.B. in Instanzvariablen vermerkter, belegter Speicher wieder freigegeben oder eine offene Datei geschlossen werden kann. Unterklassen werden `Object_Cleanup` überschreiben, um die Aufräumarbeiten ihrer Klassenebene, von denen `ClassObject` ja nichts wissen kann, durchzuführen und anschließend `Object_Cleanup` der Oberklasse aufzurufen. `Object_Delete` gibt nur den durch die Objektstruktur belegten Speicher frei und darf bei Objekten, die nicht über `CreateObject` oder `CreateObjectNoClear` erzeugt wurden, keinesfalls aufgerufen werden. Dies trifft z.B. auf Objekte zu, die auf dem Stack reserviert wurden.

### 7.2.2 Erzeugen eines Objektes

#### Objekt-Speicherplatz

Ein Objekt kann prinzipiell auf zwei Arten an Speicherplatz kommen. Die erste Art ist der Aufruf `CreateObject`, der den Speicher per Systemfunktion *AllocMem()* belegt und alle Instanzvariablen des Objekts löscht. Zu dieser Art gehört auch `CreateObjectNoClear`, das wie `CreateObject` arbeitet, aber die Instanzvariablen nicht löscht. Der zweite Weg besteht darin, den Speicherplatz selber zu reservieren. Die Größe eines Objektes ist ja im allgemeinen durch die Definition des Objektes bekannt bzw. sollte über Includefiles leicht erreichbar sein. Damit die Klasse des Objekts bekannt ist, muß aber als erstes der Makro `INITOBJECT` aufgerufen werden. Danach ist das Objekt im selben Status wie nach der Erzeugung durch `CreateObjectNoClear`, d.h., eventuell wird es noch nötig sein, die Instanzvariablen zu löschen. Geeigneter Speicherplatz könnte sich u.a. auf dem Stack befinden.

#### Objekt-Initialisierung

Das Anlegen des Objektes reicht in der Regel nicht aus. Die meisten Objekte brauchen einen Anfangszustand für die Instanzvariablen. Deswegen existiert die Methode `Object_Initialize`, die in Unterklassen überschrieben werden sollte, damit die dortigen Initialisierungen wie Belegen von zusätzlichem Speicher durchgeführt werden können. Beispielsweise wird die Methode für die Klasse `ClassWindow` dann `Window_Initialize` heißen (siehe auch 7.5 für Namensregelung).



Nach der Initialisierung ist das Objekt fertig zum Gebrauch entsprechend der vorgesehenen Methoden.

### 7.2.3 Methodenaufrufe

Ein normaler Methodenaufruf wird mit Hilfe der Direktive METHOD vorgenommen. Das erste Argument der Direktive ist der Zeiger auf das Objekt, das zweite die Methoden-ID, z.B. `Window_Initialize`. Der Objektzeiger darf dabei jede beliebige Adressierungsart sein. Die Argumente, die die Methode bekommen soll, werden wie bei normalen Unterroutinenaufrufen entweder über den Stack oder in Registern übergeben. Für die durch `ClassObject` vordefinierten Methoden sollte unbedingt die Registerbelegung eingehalten werden, damit keine Fehler durch Unachtsamkeit entstehen. Ein Beispiel für Methodenaufrufe:

```
        ; A5 zeige auf Objekt "Benji" der Klasse "Hund"
        METHOD  a5,Hund_Belle
        METHOD  tom,Katze_AufDenBaum
        METHOD  #jerry,Maus_Lachen
        ...
tom     ds.l    1       ; Zeiger auf Objekt der Klasse "Katze"
jerry   ds.b    Maus_SIZEOF
```

*Nach der METHOD-Direktive ist das Register A6 verändert!* Innerhalb der aufgerufenen Methode zeigt das Register A5 auf das Objekt, auf das die Methode angewandt wird, und nimmt somit die Stellung von *Self* ein. Dieses Vorbesetzen von A5 erledigt die METHOD-Direktive.

### 7.2.4 Vernichtung von Objekten

Nach Gebrauch muß ein Objekt früher oder später vernichtet werden. Entsprechend der Erzeugung des Objektes muß zuerst die in Unterklassen überschriebene Methode `Object_Cleanup` aufgerufen werden (z.B. `Window_Cleanup` für Objekte der Klasse `Window`). Diese wird die Initialisierung rückgängig machen, die in der Initialize-Methode vorgenommen wurde. Die weitere Vorgehensweise hängt von der Speicherplatzart ab, die das Objekt belegt. Falls es per `CreateObject` oder `CreateObjectNoClear` belegt wurde, muß zum Freigeben noch seine Delete-Methode aufgerufen werden (z.B. `Window_Delete`. Für selbst reservierte Objekte, die den `INITOBJECT`-Aufruf erfuhren, ist von Seiten des Objektsystems her nichts mehr zu tun. Nach der Vernichtung dürfen keinesfalls noch Methoden an das Objekt geschickt werden.

## 7.3 Implementierung von Klassen

Um eigene Klassen definieren zu können, stellt OMA spezielle Makros in dem Includefile ooma.i zur Verfügung. ooma.i inkludiert automatisch das Amiga-Includefile exec/types.i, damit Strukturdeklarationen wie im folgenden einfach möglich sind.

### 7.3.1 Schritt 1: Objektstruktur deklarieren

Die Objektstruktur muß zuerst deklariert werden, da die Größe eines Objektes für die Klassendeklaration bekannt sein muß. In die Objektstruktur geht auch ein, von welcher Klasse geerbt wird. Dies geschieht einfach, indem die Instanzvariablen der Oberklasse am Anfang der Objektstruktur stehen und die neuen dahinter kommen, z.B. Erben von `ClassObject`:

```
STRUCTURE Rechteck,object_SIZEOF
UWORD  rechteck_Höhe
UWORD  rechteck_Breite
LABEL  rechteck_SIZEOF          ; Größe eines Rechtecks
```

`object_SIZEOF` ist dabei die Anzahl von Bytes, die ein Objekt der Klasse `ClassObject` benötigt, und stammt aus dem Includefile ooma.i. Dadurch, daß die Instanzvariablen der Oberklasse in allen Unterklassen denselben Abstand vom Beginn des Objektes haben, wird Vererbung überhaupt erst möglich.

### 7.3.2 Schritt 2: Methoden deklarieren

Nach der Objektstruktur werden die Methoden deklariert. Obwohl es möglich ist, die Methoden der Oberklasse, die nicht überschrieben werden sollen, mit dem Präfix der Oberklasse anzusprechen, schafft es mehr Übersicht, diese Methoden mit dem Präfix der neuen Klasse zu deklarieren. Dann braucht man sich nämlich nicht mehr beim Methodenaufruf darum zu kümmern, zu welcher Oberklasse die Methode gehört (dies ist für den Aufruf ja unerheblich). Diese Neudeklaration geschieht mittels des `SAMEMETHOD`-Makros:

```
SAMEMETHOD      Rechteck_Initialize,Object_Initialize
SAMEMETHOD      Rechteck_Cleanup,Object_Cleanup
SAMEMETHOD      Rechteck_Delete,Object_Delete
```

Die neuen Methoden-IDs, die erst in der neuen Klasse deklariert werden, müssen direkt an die Methoden-IDs der Oberklasse anknüpfen. Dazu gibt es den Makro `BEGINMETHOD`, der als Argument die Klassencontainergröße der Oberklasse braucht. Die neuen Methoden-IDs werden dann mit dem Makro `DEFINEMETHOD` deklariert. Am Ende muß die Klassencontainergröße unserer neuen Klasse ermittelt werden, damit andere Klassen später von uns erben können. Dies geschieht per `ENDMETHOD`-Makro und wird später von `BEGINMETHOD` so ausgenutzt, wie wir es gerade selber gemacht haben. Beispiel:

```
BEGINMETHOD     Object_ClassSize
DEFINEMETHOD    Rechteck_SetzeHöhe
DEFINEMETHOD    Rechteck_SetzeBreite
DEFINEMETHOD    Rechteck_BerechneUmfang
DEFINEMETHOD    Rechteck_BerechneFläche
ENDMETHOD       Rechteck_ClassSize
```



### 7.3.3 Schritt 3: Klassencontainer anlegen

Die beiden vorigen Schritte sollten in einem Includefile vorhanden sein, damit Benutzer der Klasse die Methodennamen einfach inkludieren können. In dem Modul, das die Klasse und die Methoden schließlich implementiert, muß Platz für den Klassencontainer angelegt werden. Mit dem Makro `DEFINECLASS` und den Argumenten Klassenname und Klassencontainergröße läßt sich dies einfach bewerkstelligen:

```
CNOP 0,4
DEFINECLASS     ClassRechteck,Rechteck_ClassSize
```

Wegen makrointerner Verwaltung ist es wichtig, daß die Klassendeklaration im Quelltext vor den Methoden geschrieben wird. Der Platz für den Klassencontainer kann beliebig in einem Hunk liegen, sollte allerdings vorzugsweise in einen BSS-Hunk und auf langwort-ausgerichtete Adresse kommen. Daher das obige CNOP. Der Name der Klasse, hier `ClassRechteck`, sollte zum Erzeugen von Objekten mit `CreateObject` per XDEF-Direktive exportiert werden.

Sollten Sie noch eigene Daten mit einer Klasse assoziieren können, so verwenden Sie dafür das reservierte Feld `APTR class_pUserData` im Klassencontainer, z.B. `move.l a0,ClassRechteck+class_pUserData`.

### 7.3.4 Schritt 4: Methoden implementieren

#### Einfache Implementierung

Methoden werden prinzipiell genau so wie normale Unterroutinen geschrieben. Dies gilt auch für die Parameterübergabe, die in Registern (außer A6, s.o.) oder auf dem Stack stattfinden kann. Der Unterschied besteht darin, daß der Einsprungpunkt der Methode mit dem Makro `METHODBODY` festgelegt wird, da die die Namen der in Schritt 2 deklarierten Methoden-IDs ja schon vergeben sind. Das Label, das einer Methode zugewiesen wird, ist die Methoden-ID mit einem »m_«-Präfix, z.B. `m_Rechteck_SetzeHöhe`. Innerhalb einer Methode ist das Register A5 der Self-Zeiger auf das Objekt und ermöglicht schnellen Zugriff auf die Instanzvariablen. Beispiel:

```
; Setze die Höhe eines Rechtecks.
; Eingabe:  A5: Self.
;           D0: neue Höhe.
; Ausgabe:  Keine.

METHODBODY      Rechteck_SetzeHöhe
        move    d0,rechteck_Höhe(a5)
        rts
```

Um in den Methoden bequemer programmieren zu können, sollten Sie die Möglichkeiten der Makros im Includefile macros.i ausnutzen. Dort finden Sie das automatische Retten von benutzten Registern und das Anlegen von lokalen Variablen auf dem Stack.

Natürlich kann von einer Methode aus eine andere Methode u.U. eines anderes Objekts aufgerufen werden. Ein Problem taucht aber auf, wenn Sie in einer überschriebenen Methode ihrer Klasse die Methode des gleichen Namens der Oberklasse aufrufen wollen. Das normale Verwenden von METHOD würde dabei nur zum erneuten Aufruf der Methode derselben Klassenebene und damit wahrscheinlich zu einer Endlosschleife führen. Daher gibt es die Direktive SMETHOD, die den gewünschten Effekt bewirkt. Aus Komfortgründen sollten Sie aber den Makro SUPER aus ooma.i verwenden, der die gleichen Argumente wie METHOD hat.

### Aufrufketten

Überschriebene Methoden, die mit den Methoden gleichen Namens in der Oberklasse interagieren, bilden über die Klassenebenen hinweg Aufrufketten. Ein gutes Beispiel hierfür sind Methoden, die Object_Initialize und Object_Cleanup oder eine ihrer Folgemethoden überschreiben. Hier muß besonders auf die Reihenfolge der SUPER-Aufrufe geachtet werden, damit das System im konsistenten Zustand verbleibt. Beim Initialisieren sollten zuerst immer die Instanzvariablen der Oberklassen initialisiert werden, erst dann die eigenen:

```
METHODBODY      Window_Initialize
        PROC    d1/a0-a1/a6
        ; Erst Oberklasse initialisieren.
        SUPER   a5,Window_Initialize
        ; Belege privat genutzten Speicher.
        clr     window_width(a5)
        clr     window_height(a5)
        moveq   #16,d0
        moveq   #0,d1
        CALL    AllocMem,_SysBase
        move.l  d0,window_myMemory(a5)
        ENDPROC
```

Aus Gründen der Übersichtlichkeit wird hier keine Prüfung durchgeführt, ob die Initialisierung der Oberklasse gelang oder ob der Speicher auch belegt werden konnte. Beim Aufräumen kurz vor dem Löschen des Objektes sollten zuerst die eigenen Arbeiten erledigt werden, danach die der Oberklassen:

```
METHODBODY      Window_Cleanup
        PROC    d0-d1/a0-a1/a6
        ; Gib privaten Speicher wieder frei.
        move.l  window_myMemory(a5),d0
        beq.s   .nomem          ; es gab keinen Speicher
        move.l  d0,a1
        moveq   #16,d0
        CALL    FreeMem,_SysBase
```



```
        ; Jetzt Oberklasse aufräumen.
.nomem  SUPER   a5,Window_Cleanup
        ENDPROC
```

### 7.3.5 Schritt 5: Klassen registrieren

Damit das Objektsystem überhaupt Kenntnis von der Klasse hat, muß die Klasse vor jeglicher Verwendung bei ihm angemeldet werden. Dafür gibt es im Objektmodul die Routine RegisterClass, die von einer Initialisierungsroutine in Ihrem Klassenmodul aus aufgerufen werden muß. Ein Argument dieser Routine ist ein Zeiger auf ein Initialisierungsfeld. Dieses Feld legt fest, welche Methoden in der Klasse neu vorhanden sind. Es wird per Makro OVERWRITE deklariert und muß immer mit dem ENDOVERWRITE-Makro abgeschlossen werden. Wird eine neue Methode der Klasse hier vergessen, dann ist entweder der Methodenzeiger der Oberklasse oder ein Nullzeiger in der Methodentabelle eingetragen. Dies führt dann beim Methodenaufruf zu einem Fehler. Beispiel:

```
MeldeRechteckAn
        PROC    d0-d1/a0-a2
        lea     ClassRechteck,a0        ; Klassenzeiger
        lea     ClassObject,a1          ; Oberklasse
        lea     .array,a2               ; Initialisierungsfeld
        moveq   #rechteck_SIZEOF,d0     ; Objektgröße
        moveq   #Rechteck_ClassSize,d1  ; Klassengröße
        jsr     RegisterClass
        ENDPROC

.array  OVERWRITE       Rechteck_Initialize
        OVERWRITE       Rechteck_SetzeHöhe
        OVERWRITE       Rechteck_SetzeBreite
        OVERWRITE       Rechteck_BerechneUmfang
        OVERWRITE       Rechteck_BerechneFläche
        ENDOVERWRITE    ; Abschluß
```

Wenn die Initialisierungsroutinen aller Klassen aufgerufen worden sind und damit alle Klassen angemeldet wurden, muß noch die Routine InitializeClasses im Objektmodul in Aktion treten. Sie initialisiert alle Klassencontainer, trägt die Methodentabellen ein und bringt somit das Objektsystem in den Anfangszustand. Erst ab diesem Zeitpunkt dürfen Objekte erzeugt und verwendet werden.

## 7.4 OOP-Beispiel

Sie finden auf der Originaldiskette ein Beispiel für OOP mit dem OMA mit ausführlichen Kommentaren. Das Beispiel zeigt die Deklaration zweier Klassen ClassRechteck und ClassQuadrat, wobei Quadrate von Rechtecken erben, die wiederum ClassObject

als Oberklasse besitzen. In dem Beispiel kommen alle wichtigen Techniken vor, die Sie für OOP mit dem OMA brauchen, u.a. auch das Belegen von Objekten per `CreateObject` und auf dem Stack.

## 7.5 Benennungskonventionen

Damit einigermaßen Klarheit über Klassen, Methoden etc. herrscht, gibt es bei OMA eine Konvention für die Benennung.

- Alle Klassen beginnen mit dem Präfix »Class«, also `ClassObject` und `ClassWindow`.
- Methodennamen beginnen mit dem Klassennamen ohne »Class«-Präfix, aber mit einem folgenden Unterstrich, beispielsweise `Object_Initialize`, `Object_Cleanup` und `Window_Close`.
- Instanzvariablen sollten wie Methodennamen beginnen, nur sollte der Klassenname darin klein geschrieben werden, z.B. `window_width`.
- Damit die Größe eines Objektes zum Erzeugen der Klasse bekannt ist, sollte am Ende der Objektdefinition immer ein Label mit dem Suffix »_SIZEOF« deklariert werden, z.B. `window_SIZEOF`.
- Entsprechend wird für die Methodendeklaration die Größe des Klassencontainers benötigt, welche die Form des Klassennamens mit dem Suffix »_ClassSize« haben sollte, z.B. `Window_ClassSize`.

Diese Konvention sollte von allen befolgt werden.

# Stichworte


_RS ..... 99, 103

Abhängigkeitszeile ..... 58
ABSOLUTE ..... **92**
ACTIVATEDOC ..... 38, **44**
ACTIVATELASTDOC ..... **44**
ACTIVATENEXTDOC ..... **44**
ACTIVATEPREVDOC ..... **45**
ACTIVATEWINDOW ..... **45**
Adressierungsarten ..... 85
AINC: ..... 75, 76, 95
amiga.lib ..... 172
Anhalten des Asssemblers ..... 78
APPENDLINE ..... **53**, 54
AppWindows ..... 16
ASSEMBLE ..... 28, 56, **56**, 60, 63
Assembler ..... 28, 56–58, 60, 63
Assembliergeschwindigkeit ..... 72
Aufruf des Assemblers ..... 74
Ausdrücke ..... 81
Ausführbare Programme .. 74, 77, 96, 100, 101, 119

BACKSPACE ..... 29, **54**
Backspace-Taste ..... 29
Backup ..... 36, 40, 63
BASEREG ..... 93, **93**, 125, 126
Basereg-Modus . 93, 94, 125, 127, 154
Basisregister ..... 93, 127
BEEPSCREEN ..... **45**
BLK ..... **102**
Blockbefehle ..... 39
Blocklegen ..... 23, 24
Breakpoint ... 146, 148, 150, 154, 155
BSS ... 91, **92**, 93, 104, 121, 125–127
Bss ..... 121
Bss-Sektion ..... 92–94, 121, 125–127

CAAR ..... 88
CACR ..... 88
Case-Unterschied ..... 73, 76, 77, 81, 85, 91, 93, 96, 98, 99, 108, 111, 115, 119
CCR ..... 88
CHANGEWINDOW ..... **45**
CLEARBOOKMARK ..... **44**
CLEARERRORLIST ..... **57**, 61
CLEARFILE ..... **38**
CLEARSAVEDATE ..... **37**
CLEARSOL ..... **55**
CLI ..... 74–76
Clipboard ..... 39
CLIST ..... 76, **106**, 121, 132
CLOSEFILE ..... **38**
CNOP ..... 93, **93**, 183
CODE . 91, **92**, 93, 104, 121, 125–127
Code ..... 121
COMMENT ..... **98**, 119, 121
CONV2LOWER ..... **55**
CONV2SPACES ..... **40**
CONV2TABS ..... **40**
CONV2UPPER ..... **55**
COPYBLK ..... **39**
COPYRIGHT ..... 34, **51**
CPID ..... **91**
Cross-Referenz ..... 77, 128, 129
CRP ..... 88
CSTRING ..... 118
CURRENTERROR ..... **57**
CURSOR ..... **41**, 62
Cursortasten ..... 30
CUTBLK ..... **39**

DATA ..... 91, **92**, 104, 121
DB ..... **102**

DBug.memory ..... 143, 159
DBug.prefs ..... 136
DC ..... 81, **101**, **102**, 112–116, 118
DCB ..... **102**, 112, 129
DD ..... **102**
Default-Includedirectory ..... 76, 95
DEFINEKEY ..... **49**
DELETE ..... 29, 54, **54**
Delete-Taste ..... 29
DELETEBLK ..... **39**
DELETEEOL ..... **54**
DELETEEOW ..... **55**
DELETELINE ..... 21, 53, **53**
DELETESOL ..... **54**
DELETESOW ..... **55**
DELETEVIEW ..... 19, **45**
Devpac ..... 71, 72, 79, 118, 119
DFC ..... 88
Disassembler ..... 142, 160
DL ..... **102**
DLink ..... 71, 126
DP ..... **102**
Drucken ..... 28, 38
DS ... 75, 93, **102**, 104, 121, 125–127, 129
DTT0 ..... 88
DTT1 ..... 88
DW ..... **102**
DX ..... **102**

Editierbereiche ..... 18
Editor ..... 71, 74, 79
Einfügemodus ..... 21
ELSE ..... 109, **109**, 114, 121, 129
ELSEIF ..... **109**
END . 94, **94**, 104, 121, 125 127, 131
ENDC .. 107–109, **109**, 111–114, 116, 121
ENDIF ..... **109**
ENDM .. 99, 110, **110**, 111–114, 116, 118, 121
ENDMAC ..... 99, **110**
ENDMACRO ..... 48, **48**
ENDR ..... **97**, 113, 121
ENDSRC ..... **94**
Enter-Taste ..... 29
ENTRY ..... **100**

EPROM-Programme ..... 92
EQU 75, 79, 81, 98, **98**, 103, 121, 129, 131
EQUATE ..... **98**
Equatefile ..... 74
EQUR . 75, 85, 99, **99**, 119, 121, 127, 129, 131
EQURL ..... **100**
EVEN ..... **93**
EXCHANGECHARS ..... **54**
EXECAREXXMACRO ... 16, 35, 48, **48**, 63
EXECAREXXSTRING ... 48, **48**, 63
Expandieren ..... 40
EXPORT ..... **100**, 101
EXTERN ..... **101**
EXTRN ..... **101**

FAIL ..... **98**, 111, 112, 116, 121
Fast-Modus ..... 74–78
Fehlermeldungen ..... 77, 124
FEQU ..... 83, 99, **99**, 121, 131
FETCHOPENSIZE ..... **47**, 65
Filerequester ..... 16, 26, 36, 37, 40, 47–49, 56, 57, 63, 66
FIND ..... 32, **50**
FINDCHANGE ..... **50**
FIRSTERROR ..... **56**, 60
FORMAT ..... **98**, 119, 121
FORMATPARAGRAPH ..... 22, **53**
FP0 ..... 88
FP1 ..... 88
FP2 ..... 88
FP3 ..... 88
FP4 ..... 88
FP5 ..... 88
FP6 ..... 88
FP7 ..... 88
FPCR ..... 88
FPIAR ..... 88
FPSR ..... 88
FSET ..... 83, **99**, 121, 131

GET ..... **95**
GETBLK ..... **41**
GETBLKINFO ..... **41**
GETCHAR ..... **51**



GETCURSORPOS ..... **51**
GETDOCUMENTS ..... **51**
GETFILEINFO ..... **52**
GETFILEPATH ..... **52**
GETLINE ..... **51**
GETPORT ..... **52**
GETPREFS ..... **47**, 62
GETPRIORITY ..... **50**
GETSCREENINFO ..... **52**
GETVERSION ..... **51**
GETVIEWINFO ..... **52**
GETWINDOWINFO ..... **52**
GETWORD ..... **51**
GOTOBOOKMARK ..... 44, **44**
GOTOCOLUMN ..... 41, **41**
GOTOLEFTMARGIN ..... **41**
GOTOLINE ..... 41, **41**
GOTORIGHTMARGIN ..... **41**
GROWVIEW ..... 19, **45**

hartes Leerzeichen ..... 22, 33
Headerfile ..... 74
HELP ..... **51**
Help-Taste ..... 25, 27, 29, 31, 32
HIDEFILE ..... 20, 26, **38**, 52
History ..... 146, 155
HiTex-Editor ..... 71, 79
HiTex.default.info ..... 17, 20
HiTex.hit ..... 16, 35, 46
HiTex.prefs ..... 15, 16, 29, 64
HiTex.setup ..... 17, 67

IBYTES ..... **95**
Icon ..... 16, 17, 19, 28, 36, 40, 62
IDENTIFY ..... **100**
IDNT ..... 81, **100**, 121
IFC ..... **108**, 114
IFD ..... 78, **108**, 109
IFEQ ..... 107, **108**, 109, 111
IFGE ..... **109**
IFGT ..... **109**
IFLE ..... **109**, 111
IFLT ..... **109**
IFNC ..... **108**, 113, 116
IFND ..... 78, **108**
IFNE ..... 108, **108**, 109, 112, 116
ILLEGAL ..... 90

IMPORT ..... 101, **101**
INCBIN ..... **95**, 119, 129
INCDIR ..... 94, 95, **95**, 121
INCLUDE 74, 75, 81, 94, **94**, 121, 131
INCPRE ..... 75, 95, **95**, 131
Inkompatibilität ..... 92, 118, 119
INSERT ..... **53**
INSERTASCII ..... 22, **54**
INSERTSPACE ..... 29, **54**
ISP ..... 88
ITT0 ..... 88
ITT1 ..... 88

Klammersuche ..... 51
Kommandozeile, Make ..... 58
Komprimieren ..... 40
Konstanten ... 79, 81, 82, 91, 94, 102, 131

LEFT ..... 115, **115**, 116
Linkbare Programme 77, 96, 100, 101, 119
Linker ..... 71, 94, 100, 101, 126
LIST ..... **106**, 119, 121, 132
LISTCHAR ..... **107**
Listingfile 74, 76, 77, 96, 105, 128, 129
LLEN ..... **106**, 119, 121, 130
LOADASSEMBLER ..... **56**, 60, 63
Lokale Symbole/Label 76, 80, 81, 96, 118
LOWERCASE ..... 40, **40**, **55**

MACRO . 99, 110, **110**, 111–114, 116, 118, 121, 131
MAKE ..... **38**, 56, 58, 59
Makefile ..... 38, 58
Makro ... 74, 75, 79, 90, 98, 104, 105, 110, 120, 130, 131
Makroaufnahme ..... 48, 110
Makroaufruf ..... 120
Makroausführung ..... 49
Makroexpansion ... 76, 107, 110, 111, 115, 119, 128
Marke ..... 20, 23
MASK2 ..... **98**, 121
MC68000 ..... **90**, 102, 125
MC68010 ..... **90**, 102

MC68020 ..... **90**
MC68030 ..... **90**
MC68040 ..... **90**
MC68881 ..... **90**
MC68882 ..... **90**
Mehrpass-Modus .. 72, 75, 77, 93, 122
Meldungsfile ..... 74
MetaComCo ..... 71, 72, 79, 92, 118
METHOD ..... **104**, 181
Methoden-ID ..... 104, 179, 181
MEXIT ..... 99, 110, **110**, 111, 121
MID ..... **115**, 116
MLIST ..... 76, 96, **107**, 121
MMUSR ..... 88
MOVE ..... **41**
MOVEBOOKMARK ..... **44**
MOVEDOWN ..... **42**
MOVEDOWNSCR ..... **43**
MOVEEOF ..... **43**
MOVEEOL ..... **43**
MOVELEFT ..... **42**
MOVEMATCHBKT ..... 23, **51**
MOVENEXTTABSTOP ..... **43**
MOVENEXTWORD ..... 42, **43**
MOVEPREVTABSTOP ..... **43**
MOVEPREVWORD ..... **43**
MOVERIGHT ..... **43**
MOVESOF ..... **43**
MOVESOL ..... **43**
MOVEUP ..... **42**
MOVEUPSCR ..... **43**
MSP ..... 88

NARG ..... 99, 111, 116
NEWFILE ..... **36**, 63
NEWLINE ..... 29, 53, **53**, 62
NEWLINEINDENT ..... 29, **53**
NEXT ..... 30, **42**, 43, 62
NEXTERROR ..... **56**, 57, 60
NEXTVIEW ..... 19, **45**
NOCLIST ..... 76, **107**, 121
NOFORMAT ..... **98**, 121
NOL ..... **106**
NOLIST ..... **106**, 121, 132
NOMLIST ..... 76, 96, **107**, 121
NOOBJ ..... 76, **97**, 121
NOOPTIM ..... 76, 96, **97**, 120, 121
NOPAGE ..... 76, **105**, 106, 121, 128

Objekt ..... 104
Objektfile ... 74, 76, 95–97, 100, 101, 119, 121, 131, 132
Objektfilename ..... 95
OFFSET .. 75, 82, 93, 103, **103**, 104, 129
OOP ..... 72, 104
Opcode ..... 79, 90, 110
OPENASSEMBLE 28, 56, **56**, 60, 63
OPENBOOKMARKS ..... 26, **43**
OPENBUFFERLIST ..... 26, **36**
OPENCLIP ..... **39**
OPENFILE ..... 26, 35, **36**
OPENFIND ..... 31, **50**
OPENNEWFILE ..... 20, **36**, 60, 63
OPENPREFS ..... **47**, 62
OPENPREFWINDOW ..... **46**, 62
OPENSCREENPREFWINDOW **47**, 64
Operatoren ..... 82, 119
OPT .... 76, 77, 80, 96, **96**, 100, 101, 119–121
Optimierender Makro-Assembler . 13, 22, 56, 60, 61, 63, 148, 154, 163
Optimierung ..... 76, 77, 96, 97, 122
OPTIMIZE ..... 96, **97**, 120, 121
ORG ..... **92**, 104, 119, 121
ORIGIN ..... **92**
OUTPUT ..... 74, 95, **95**, 96, 121
Overstrike ..... 18, 21, 29, 53, 54

PAGE ..... 76, **105**, 121, 128
PAGEUP ..... **105**
PASTE ..... 30, **39**, 40
PLAYMACRO ..... 48, **49**
PLEN ..... **106**, 121
POSITION ..... 30, 42, **42**
Preferencedatei .... 15, 17, 47, 56, 65
Preferences . 15, 17–22, 27, 28, 37, 47, 48, 51, 56, 62
PREVERROR ..... **56**
PREVIOUS ..... 30, **42**, 43, 62
PREVVIEW ..... 19, **45**
PRINTFILE ..... **38**



PSR ..... 88
PSTRING ..... 118

QUIT ..... 37, **38**

Rand, linker ..... 22, 29, 41, 53, 62
Rand, rechter ..... 22, 29, 41, 62
READERRORLIST ..... **57**, 61
RECORDMACRO ..... 48, **48**
REG ..... 100, **100**, 121, 129, 131
Registermaske ..... 100, 130
Registersymbol ..... 85, 99, 130
Registersymbole ..... 119
REPLACEWORD ..... **55**
REPT ..... **97**, 112, 113, 121
REQUESTBOOL ..... **49**
REQUESTFILE ..... **49**
REQUESTNOTIFY ..... **49**
REQUESTNUM ..... **49**
REQUESTSTR ..... **49**
RESERVE ..... **103**
Return-Taste 25, 27, 29, 31, 32, 59, 62
RIGHT ..... **115**
RORG ..... **92**, 104, 118
RORIGIN ..... **92**
RS ..... 75, 103, **103**, 104, 121, 129
RSRESET ..... **103**
RSSET ..... **103**
RUNPROGRAM ..... **56**, 61, 63
RX ..... **48**
RXS ..... **48**

SAVEBLK ..... **40**
SAVEFILE ..... **36**, 37, 38, 40, 63
SAVEFILEAS ..... 27, **37**
SAVEFILEQUIT ..... **37**
SAVEMACRO ..... 27, 48, **49**
SAVEMODIFIEDFILES ..... **37**, 38
SAVEPREFS ..... 26, **47**, 62, 65
SAVEWINDOWINFO ..... 46, **46**
SCREEN2BACK ..... **46**
SCREEN2FRONT ..... **46**
SCROLL ..... **41**
Scrolling ..... 21, 24
SECTION ... 86, 91, **91**, 92, 104, 121
Seitenlänge ..... 106
Seitenvorschub ..... 76, 105, 106, 128
Sektionstyp ..... 91
SELECTALL ..... 23, **41**
SELECTLINE ..... 23, **40**
SELECTWORD ..... 23, **40**
SET .. 98, **98**, 99, 100, 110, 112, 116, 121, 129, 131
SETBOOKMARK ..... **44**
SETCHAR ..... **54**
SETCURRENTDIR ..... **37**
SETFILEPATH ..... **37**
SETHOMEDIR ..... 27, 28, **37**
SETPREFS 22, 29, 41, 43, 47, **47**, 54, 62, 70
SETPRIORITY ..... **50**
SETR ..... 85, 99, **99**, 121, 129, 131
SETREG ..... **100**, 129, 131
SETRL ..... **100**
SETSAVEDATE ..... 37, **37**
SETVAL ..... **99**
SFC ..... 88
SHIFTLEFT ..... **39**
SHIFTRIGHT ..... **39**
SHRINKVIEW ..... 19, **45**
Simulationsmakros ..... 118
SMETHOD ..... 104, **104**, 184
SNAPSHOT ..... 20, **38**
Source-Level-Debugging . 72, 76, 132, 148, 163
SP ..... 88
SPC ..... **105**, 121
Speichertyp ..... 91, 92
SPLITVIEW ..... 18, **45**
SR ..... 88
SRP ..... 88
Standardendung ..... 74
Startoptionen ..... 74
Statuszeile ..... 18, 19, 50, 60
Stringregeln, Übliche ..... 73
STRLEN ..... **115**
SUBTTL ..... **98**, 119, 121
SUPER ..... 105
SWAPCASE ..... 40, **40**, **55**
SWAPCHARS ..... **54**
Symbollänge ..... 72
Symbolsyntax ..... 81
Symboltabelle ..... 76–78, 96, 120, 128–130

Tabellenfile ..... 75
TABINDENT ..... **53**
Tabulator-Taste .... 22, 25, 27, 30–32
Tabulatorexpansion ..... 21
Tastaturbelegung ..... 17, 29, 68
TC ..... 88
Terminologie ..... 73
TEXT ..... **53**
TITLE ..... **106**
Titlezeile ..... 18
TOGGLECHARCASE ..... **54**
Tooltypes ..... 16, 20, 62
Trennzeile ..... 18
TT0 ..... 88
TT1 ..... 88
TTL ..... 81, **106**, 121, 128

UNDELETELINE ..... 53, **53**
Undo ..... 54
UNDOLINE ..... **54**
UNLOADASSEMBLER ..... **56**, 61
UNMARKBLK ..... **39**
UNSNAPSHOT ..... 20, **38**
UPPERCASE ..... 40, **40**, **55**
URP ..... 88
USP ..... 88

VALOF ..... 115, **115**, 120
VBR ..... 88
Videomodus ..... 64
Vorwärtsreferenz ..... 93, 98, 122

WB2FRONT ..... **46**
Wildcards ..... 31, 32
WINDOW2BACK ..... **46**
WINDOW2FRONT ..... **46**
Wordwrap . 18, 20, 22, 29, 42, 53, 54, 62
WRITECHARCODE ..... **49**
WRITESTRING ..... **50**

XDEF .... 77, 96, 100, **100**, 119, 121, 125–127, 183
XREF 77, 81, 96, 100, 101, **101**, 119, 121, 125–127

Zeilenende ..... 79
Zeilenlänge ..... 72, 106
Zeilensyntax ..... 79